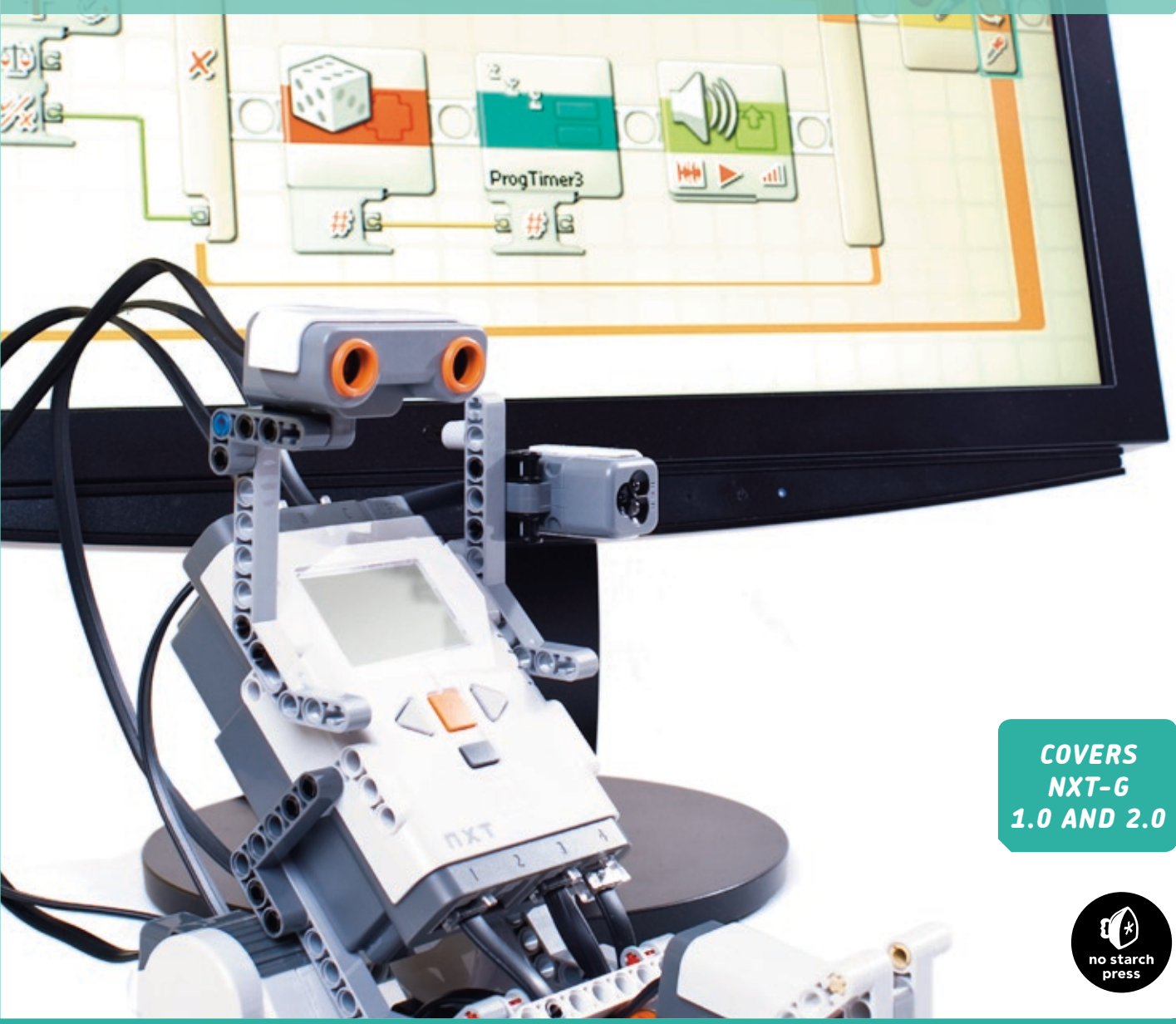
THE ART OF LEGO® MINDSTORMS® NXT-G PROGRAMMING

terry **griffin**



COVERS NXT-G 1.0 AND 2.0

no starch press

THE ART OF LEGO® MINDSTORMS® NXT-G PROGRAMMING

THE ART OF LEGO® MINDSTORMS® NXT-G PROGRAMMING

terry **griffin**

14 13 12 11 10 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-218-9
ISBN-13: 978-1-59327-218-0

Publisher: William Pollock
Production Editor: Megan Dunchak
Cover and Interior Design: Octopod Studios
Technical Reviewer: Damien Kee
Copyeditor: Kim Wimpsett
Compositor: Lynn L'Heureux
Proofreader: Nancy Sixsmith

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
38 Ringold Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Griffin, Terry, 1962-
The art of LEGO Mindstorms NXT-G programming / Terry Griffin.
p. cm.
Includes index.
ISBN-13: 978-1-59327-218-0
ISBN-10: 1-59327-218-9
1. Robots--Design and construction. 2. Robots--Programming. 3. Lego Mindstorms toys. I. Title.
TJ211.G75 2010
629.8'9252--dc22
2010017757

This book is dedicated to the love of my life, Liz, and our three wonderful daughters, Cheyenne, Sarah, and Samantha.

brief contents

acknowledgments xix
introduction xxi

chapter 1 LEGO and robots: a great combination 1
chapter 2 the NXT-G programming environment 7
chapter 3 the test robot 17
chapter 4 motion 45
chapter 5 sensors 57
chapter 6 program flow 73
chapter 7 the WallFollower program: navigating a maze 83
chapter 8 data wires 97
chapter 9 data wires and the switch block 111
chapter 10 data wires and the loop block 123
chapter 11 variables 131
chapter 12 the NXT buttons and the display block 147
chapter 13 my blocks 161
chapter 14 math and logic 179
chapter 15 files 195
chapter 16 data logging 209
chapter 17 using multiple sequence beams 221
chapter 18 the LineFollower program 233

appendix A NXT websites 253
appendix B moving from NXT-G 1.0/1.1 to NXT-G 2.0 255

index 257

contents in detail

acknowledgments ..... **xix**

introduction ..... **xxi**
who this book is for xxi
prerequisites xxi
what to expect from this book xxi
how best to use this book xxiii

1
LEGO and robots: a great combination **1**
LEGO MINDSTORMS NXT 1
the NXT online community 2
the LEGO MINDSTORMS NXT kit 2
NXT versions 3
MINDSTORMS software versions 3
art and engineering 4
qualities of a good program 4
software, firmware, and hardware 5
NXT-G 5
what you'll learn from this book 6
what's next? 6

2
the NXT-G programming environment **7**
a tour through the MINDSTORMS environment 7
a: work area 8
b: programming palettes 8
c: robo center 8
d: my portal window 8
e: configuration panel 9
f: help panel 9
g: navigation panel 9
h: controller 9
writing an NXT-G program 9
your first program 10
saving your work 10
running your program 11
your second program 11
debugging 12
reproduce the bug 12
simplify the program 12

look at the parts of the program ... 12
fix the bug ... 12
the edit-compile-test cycle ... 13
comments ... 13
adding comments ... 14
rules for working with comments ... 15
the configuration panel ... 15
general layout ... 15
changing panels ... 15
disabled items ... 16
a block's configuration icons ... 16
conclusion ... 16

3
the test robot ... 17
right-side motor ... 19
left-side motor ... 21
chassis ... 23
caster wheel ... 25
caster wheel for the NXT 2.0 retail kit ... 25
caster wheel for the original NXT retail kit and education set ... 27
attach the caster wheel ... 29
add the NXT ... 31
touch sensor bumper ... 33
attach the bumper to the chassis ... 36
ultrasonic sensor ... 37
sound sensor ... 38
color sensor or light sensor ... 39
attach the wires ... 41
the final beam ... 42
alternate placement for the color sensor ... 42
alternate placement for the ultrasonic sensor ... 43
conclusion ... 44

4
motion ... 45
the NXT motor ... 45
the move block ... 46
the move block's configuration panel ... 46
the feedback boxes ... 48
the NXT intelligent brick view menu ... 48
there and back ... 49
moving forward ... 49
turning around ... 49
testing a single block ... 50
moving back to the start ... 50
around the block ... 50
the first side and corner ... 50

the other three sides and corners ... 51
testing the program ... 52
the motor block ... 52
brake, coast, and the reset motor block ... 53
the CoastTest program ... 53
a problem with coasting ... 54
the reset motor block ... 55
the record/play block ... 55
configuration panel ... 55
the remote control tool ... 56
conclusion ... 56

5
sensors ... 57
using the sensors ... 57
the touch sensor ... 58
configuration panel ... 58
feedback box ... 58
the NXT's view menu ... 59
the BumperBot program ... 59
detecting an obstacle ... 60
backing up and turning around ... 60
testing ... 61
the sound sensor ... 61
configuration panel ... 61
setting the trigger value ... 61
BumperBot with sound ... 62
the light and color sensors ... 63
light sensor configuration panel ... 63
using the color sensor as a light sensor ... 64
the RedOrBlue program ... 64
determining red and blue values ... 64
the switch block ... 64
improving the program ... 66
using color sensor mode ... 67
the ultrasonic sensor ... 68
configuration panel ... 68
door chime ... 68
detecting a person ... 69
playing a chime ... 69
stopping the chime ... 69
the rotation sensor ... 70
configuration panel ... 70
the rotation sensor block ... 70
the BumperBot2 program ... 71
conclusion ... 72

6
program flow 73
the sequence beam 73
the switch block 73
configuration panel 74
the LineFollower program 75
more than two choices 76
using tabbed view 78
comments and tabbed view 78
the loop block 79
the keep alive block 79
the stop block 80
BumperBot3 80
conclusion 82

7
the WallFollower program: navigating a maze 83
pseudocode 83
solving a maze 86
program requirements 86
assumptions 88
initial design 88
following a straight wall 89
writing the code 89
testing 90
turning a corner 91
writing the code 91
testing 92
going through an opening 93
writing the code 94
using sound blocks for debugging 95
testing 95
final test 96
conclusion 96

8
data wires 97
what is a data wire? 97
the GentleStop program 97
tips for drawing data wires 101
the SoundMachine program 101
controlling the volume 102
using the math block 103
adding tone control to the SoundMachine program 103
understanding data types 104
using the number to text block 105
displaying the tone frequency 105
using the text block 107

adding labels to the displayed values ... 108
dealing with broken wires ... 109
conclusion ... 110

9
data wires and the switch block ... 111
the switch block's value option ... 111
rewriting the GentleStop program ... 112
advantages of using a sensor block ... 113
passing data into a switch block ... 113
passing data out of a switch block ... 113
matching more than two values ... 116
    adding and removing conditions ... 117
    the default condition ... 117
using numbers with the NXT-G 2.0 switch block ... 117
fixing the SoundMachine program's volume display ... 117
    calculating the input value using NXT-G 1.1 ... 118
    calculating the input value using NXT-G 2.0 ... 118
    modifying the program ... 118
conclusion ... 121

10
data wires and the loop block ... 123
the loop count ... 123
    creating the LoopCountTest program ... 123
    restarting a loop ... 124
    setting the final loop count value ... 124
    setting the loop condition ... 125
timers ... 125
the timer block ... 125
a programmable timer, version 1 ... 126
the compare block ... 127
a programmable timer, version 2 ... 127
a programmable timer, version 3 ... 129
conclusion ... 129

11
variables ... 131
a place for your data ... 131
managing variables ... 131
the variable block ... 132
the RedOrBlueCount program ... 133
    creating the variables ... 133
    initializing the variables ... 134
    initializing the display ... 135
    displaying the initial values ... 135
    counting the red objects ... 135
    counting the blue objects ... 137

grouping common settings ... 138
replacing long data wires with variables ... 138
the LightPointer program ... 138
defining the variables ... 139
finding the light source ... 140
initializing the values ... 140
the LightPointer program, part 1 ... 141
the LightPointer program, part 2 ... 143
constants ... 144
managing constants ... 145
the constant block ... 145
conclusion ... 146

12
the NXT buttons and the display block ... 147
the NXT buttons ... 147
the NXT button block ... 148
the PowerSetting program ... 148
defining the variable ... 148
the initial value and the loop ... 149
displaying the current value ... 149
adjusting the power value ... 150
testing the program ... 151
making the program faster ... 151
the display block ... 152
displaying an image ... 152
power setting with images ... 153
drawing on the screen ... 155
the NXTSketch program ... 155
defining the variables ... 156
initialization ... 156
drawing the line ... 156
saving the new location ... 158
testing the program ... 158
conclusion ... 159

13
my blocks ... 161
building bigger blocks ... 161
creating a my block ... 161
the custom palette ... 163
editing a my block ... 163
configuring a my block ... 164
changing the name of a configuration item ... 165
the DisplayNumber block ... 166
configuration items ... 166
controlling the line setting using a data wire ... 166

building the DisplayNumber block....167
testing....170
creating the DisplayNumber block....170
changing the names of the configuration items....171
using the DisplayNumber block....173
managing the custom palette....174
sharing programs with my blocks....175
copying files....175
create pack and go....175
advanced my block topics....175
variables and my blocks....176
nesting my blocks....176
broken my blocks....176
adding a data plug....177
conclusion....177

14
math and logic....179
computer math....179
integer math....179
range of values....179
division....180
odometer....181
floating-point math....183
range....183
precision....183
the number to text block....183
the random block....184
adding a random turn to BumperBot....184
the logic block....185
adding some logic to BumperBot....186
the range block....189
improving RedOrBlue....189
improving RedOrBlueColorMode....192
conclusion....194

15
files....195
using files....195
the file access block....195
the filename....196
the action setting....196
the type setting....196
saving the RedOrBlueCount data....197
checking for errors....199
the FileReader program....200
restoring the RedOrBlueCount data....201

managing memory 207
deleting files 207
transferring files 208
common problems 208
conclusion 208

16
data logging 209
data collection and the NXT 209
the VerifyLightPointer program 209
collecting the brightness data 210
running the program 211
analyzing the data 212
adding rotation sensor data and a timestamp 212
gaps in the data 214
setting the initial file size 215
controlling the amount of data 216
data logging using the LEGO MINDSTORMS education NXT software 2.0 217
the data-logging blocks 217
the VerifyLightPointer2 program 218
the NXT data logging application 219
conclusion 220

17
using multiple sequence beams 221
multitasking 221
adding a second sequence beam 221
avoiding a busy loop 223
adding a sequence beam to a loop block 223
the crowbar and pin technique 224
adding the sequence beam 225
expanding the loop block 226
making the light flash 227
understanding program flow rules 229
starting blocks and data wires 229
starting a loop or switch block 229
using values from a loop or switch block 229
using my blocks 230
synchronizing two sequence beams 230
the AroundTheBlock program 230
the DoorChime program 230
keeping out of trouble 232
conclusion 232

18
the LineFollower program 233
following a line 233
requirements 233
assumptions 233
the starting point 234
selecting the sensor trigger values 234
building the LineFollowerConfig program 235
testing the LineFollowerConfig program 237
changing the LineFollower program 238
improving the control algorithm 243
how far from the edge? 244
controlling the motors 248
setting the power values 248
testing the program 251
conclusion 251

A
NXT websites 253

B
moving from NXT-G 1.0/1.1 to NXT-G 2.0 255
numbers 255
block changes 255
using old programs 256
side-by-side installation 256

index 257

acknowledgments

Thanks to Bill Pollock at No Starch Press for taking a chance on a first-time author and giving me the opportunity to create this book. The patience, expertise, and professionalism of Bill, Megan Dunchak, Riley Hoffman, Ansel Staton, and the rest of the No Starch staff have made working on this project a pleasure.

Thanks to Damien Kee, who provided the technical review of this book. His knowledge of the material and insightful comments on the presentation greatly improved the end product.

Finally, thanks to my family for graciously allowing me the time to write a book and for all the help proofreading the text and testing the building instructions.

introduction

This book is about learning how to write programs for LEGO MINDSTORMS NXT robots. The LEGO MINDSTORMS software and its NXT-G programming language are powerful tools that make it easy to write custom programs. This book will teach you how to get the most out of the NXT-G language as you acquire the programming skills necessary to successfully create your own programs.

who this book is for

This book is for anyone using NXT-G to program their NXT robots, whether you're a young robotics enthusiast, an adult working with children to learn robotics, a parent, a FIRST LEGO League coach, or a teacher using NXT in the classroom. One of my goals in writing this book was to make the material accessible to young learners while going into enough depth to help students and teachers understand the hows-and-whys of NXT-G programming.

prerequisites

This book can be used with any NXT set and any version of the MINDSTORMS software. To test your programs, you'll use a single, general-purpose robot that you can build with any NXT set. There are only a few relevant differences between the NXT software versions, and I'll point them out as appropriate. Almost all the material presented in this book applies to any version of the MINDSTORMS software.

No previous programming experience is required. NXT-G is a great first programming language, and I'll explain how to use the MINDSTORMS software and each of the elements of the NXT-G language.

what to expect from this book

This book focuses on programming NXT robots, rather than on the mechanical aspects of building them. You'll learn how to work with the core parts of the NXT-G language, such as the blocks, data wires, files, and variables, and you'll see how these pieces can work together. You'll also learn some good programming practices, bad habits to avoid, and debugging strategies that will help keep your frustration level low so you can have fun while programming. You'll find numerous NXT-G programs with step-by-step instructions and explanations, as well as many small examples designed to help you understand exactly how NXT-G works.

All of the programs in this book will work with the general-purpose robot or the NXT Intelligent Brick alone. This is a book about programming NXT, not about constructing robots. As such, I've devoted as many pages as possible to in-depth coverage of the most important programming topics.

The book begins with an introduction to the NXT set and the MINDSTORMS software. This is followed by the building instructions for the test robot. The next few chapters cover the basics of the NXT-G language, culminating in a maze-solving program in Chapter 7. Chapters covering the more advanced language features follow, and the book finishes up with a sophisticated line-following program. Here's an overview of the contents of each chapter.

Chapter 1: LEGO and Robots: A Great Combination
This chapter provides a brief introduction to the LEGO MINDSTORMS NXT set and the NXT-G language. I'll also discuss the general process used for creating programs.

Chapter 2: The NXT-G Programming Environment
In this chapter you'll find an in-depth tour through the features of the MINDSTORMS software. You'll create some simple programs and learn the basic steps for debugging problems.

Chapter 3: The Test Robot
The building instructions for the test robot are given in this chapter. You'll use this general-purpose robot to test the programs in the remainder of the book.

Chapter 4: Motion
In this chapter you'll learn about the NXT motors and the NXT-G blocks that control them. You'll write several programs to learn how to make your robot move using the Move and Motor blocks.

Chapter 5: Sensors
This chapter covers the NXT sensors: the Touch, Sound, Light, Color, Ultrasonic, and Rotation Sensors. You'll learn common uses for each sensor type, as well as how to use NXT-G to control the sensors. I've also included example programs that teach you how to use each sensor.

Chapter 6: Program Flow
The Switch and Loop blocks are the main focus of this chapter. You'll learn the Switch block's various options for making decisions and how to use the Loop block to perform repeated actions. I'll also discuss the Keep Alive block and the Stop block, as they also are related to program flow. You'll use the Switch and Loop blocks, as well as some sensor and motor blocks, to write a simple line-following program.

Chapter 7: The WallFollower Program: Navigating a Maze
By the time you get to this point in the book, you'll have learned the basic features of NXT-G. This chapter walks you through the process of creating a complete program to solve a maze using a wall-following method. This chapter focuses on learning how to design, create, and debug a large program.

Chapter 8: Data Wires
Data wires are one of the most powerful features of the NXT-G language. This chapter shows you what data wires are and how to use them effectively. The discussion includes coverage of data types and introduces some blocks that are used exclusively with data wires, including the Math, Number to Text, and Text blocks. To get some practice using data wires, you'll write the SoundMachine program, which turns your robot into a sound generator.

Chapter 9: Data Wires and the Switch Block
In this chapter you'll learn how to use a data wire to control a Switch block. You'll also learn about the special rules in NXT-G for moving data between the blocks inside a Switch block and the blocks that come before or after the Switch block.

Chapter 10: Data Wires and the Loop Block
In this chapter you'll learn how to use data wires to control a Loop block. You'll learn about the NXT timers and then use the features of the Loop block to create three different programmable timers, each with its own advantages.

Chapter 11: Variables
Variables are used for storing values that your program uses. In this chapter you'll learn how to create and use variables, and you'll see how they relate to data wires. I'll also discuss constants, which are new to NXT-G 2.0.

Chapter 12: The NXT Buttons and the Display Block
In this chapter you'll learn how to use the buttons on the NXT to control your program. You'll also learn more about how to use the NXT's display screen. You'll write programs that let you enter a value into your program, using either numbers or pictures, as well as a program that turns the NXT into a sketch pad.

Chapter 13: My Blocks

A My Block is a block you create by grouping other blocks together. In this chapter you'll learn how to create a My Block, how to use My Blocks in your programs, and how to share My Blocks with other NXT users. I'll discuss how to use variables and data wires with My Blocks and how to deal with broken My Blocks.

Chapter 14: Math and Logic

This chapter covers the Math, Logic, Range, and Random blocks. A section on how to use integer math is included for NXT-G 1.0 users, as well as a section covering floating-point math for NXT-G 2.0 users.

Chapter 15: Files

In this chapter you'll learn how to use files to store information on the NXT, how to manage the NXT's memory, and how to transfer files between the NXT and your computer.

Chapter 16: Data Logging

The programs in this chapter show you how to use the NXT as a data logger. I'll cover the basics of collecting and analyzing data, as well as some common problems. The new data-logging features of the LEGO MINDSTORMS Education Software 2.0 release are also discussed.

Chapter 17: Multiple Sequence Beams

NXT-G uses multiple sequence beams to provide multitasking functionality. In this chapter you'll learn why you might want to use more than one sequence beam and how to do so effectively. You'll also learn about the ways that using multiple sequence beams can complicate a program and some tips for avoiding the most common problems.

Chapter 18: The LineFollower Program

In this chapter you'll see how to use some advanced NXT-G features to create a complex line-following program. You'll use files to configure the program settings and implement a proportional controller to create a fast and accurate line-following machine.

Appendix A: NXT Websites

This appendix contains a list of websites that provide information about NXT-G programming.

Appendix B: Moving from NXT-G 1.0/1.1 to NXT-G 2.0

This appendix discusses the changes in the NXT-G language that you'll want to be aware of when upgrading to the new MINDSTORMS software.

how best to use this book

To get the most out of this book, you should work through the step-by-step instructions for building the example programs on your computer as you are reading. Programming is a learn-by-doing activity, and you'll learn a lot more by writing and experimenting with the programs than you will by just reading about them.

The programs and accompanying discussions will make the most sense if you read the chapters in order. Several of the example programs are introduced in the early chapters and then expanded in the later chapters as you learn more about the NXT-G language. By the time you get to the end of the book, you'll have the knowledge and skills you need to be an expert NXT-G programmer.

1

LEGO and robots: a great combination

Welcome to the world of robotics. For more than 100 years, people have thought, dreamed, and written about robots. Not long ago, the only place you could find a robot was in a good science-fiction story. Today robots are very real and perform a wide variety of important jobs. Robots explore other planets, investigate deep-sea volcanoes, assemble automobiles, and perform surgery. Figure 1-1 shows one of the Mars Exploration Rovers that help scientists explore our neighboring planet. You can even buy a robot at your local department store to sweep your floors at night while you sleep.

Figure 1-1: Mars Exploration Rover (courtesy NASA/JPL)

LEGO MINDSTORMS NXT

With the LEGO MINDSTORMS NXT kit, you can build your own robot. For example, Figure 1-2 shows a simple robot you could build to explore your own space here on planet Earth. In fact, because you are using LEGO pieces, you can build lots of different robots. And although you won't find any NXT-based robots performing surgery, people have found many uses for these little robots.

Of course, the NXT kit makes a great toy, and many children enjoy creating their own robotic creations just for fun. Many adults do this too; we just call it "having a hobby" instead of "playing with toys." However, the NXT kit is more than just a toy; teachers in middle and high schools use the kits to teach science and technology. LEGO even has an education division to provide resources for teachers who are using LEGO products in the classroom.

You can also use the NXT kits to perform real science. One group recently sent nine NXT-based robots to the edge of space to perform some high-altitude experiments. Along with being small and easy to use, NXT is extremely versatile and powerful enough for scientists of all ages to use. The only limit is your imagination!

Figure 1-2: Family Room Rover

the NXT online community

A thriving online community is devoted to LEGO robotics, including websites that show hundreds of innovative robot designs. Two sites in particular, NXTasy (*http://www.NXTasy.org/*) and TheNXTStep (*http://TheNXTStep.blogspot.com/*), are well known for their message forums where users can exchange ideas and find answers to questions. These are great resources when you can't figure out why your robot isn't working the way you think it should. A quick search of the forums often provides the answer you are looking for. If you don't find a solution already posted, you can ask a question describing your particular problem. See Appendix A for a list of useful NXT-related websites.

the LEGO MINDSTORMS NXT kit

The NXT kits include LEGO pieces for building your robot, the NXT Intelligent Brick, three motors, several sensors, and the MINDSTORMS software. The exact mix of parts and sensors included depends on your version of the kit (see the following sections for more about the different versions).

The building pieces are the beam-and-pin type shown in Figure 1-3. These pieces are used in the LEGO TECHNIC product line, which includes kits to build construction equipment, aircraft, space vehicles, and race cars. These parts are both strong and lightweight, and you can connect them in a variety of ways, making them ideal for creating robots. And because the building parts are just normal LEGO pieces, you can easily use parts from other TECHNIC, BIONICLE, or even traditional block sets to create a wide variety of robotic creatures.

The NXT motors and sensors let you turn an ordinary LEGO model into a moving robot, which can react to its environment and follow your commands. The three motors provided are specifically designed to make it easy to build mobile robots using either wheels or treads. You can also use the motors to create robotic hands, cranes, catapults, and many other moving contraptions. Many robots use two of the motors to move around and the third for some other function, while some robots use the motors for other tasks and don't move around at all.

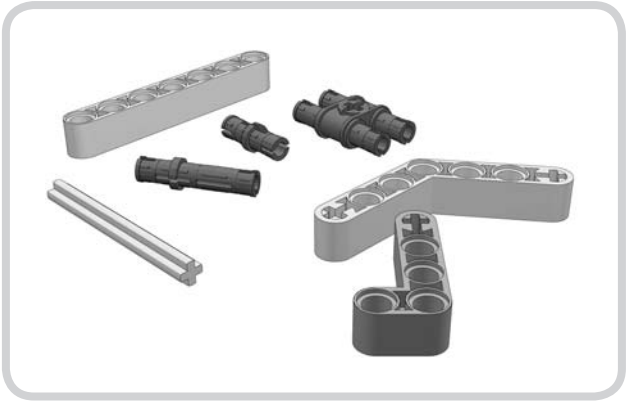


Figure 1-3: Beams and pins

LEGO makes several sensors, and your kit will come with either four or five, depending on which version you have. The sensors included in the NXT kits are the Ultrasonic, Touch, Light, Color, and Sound Sensors.

- The Ultrasonic Sensor measures the distance to an object or obstacle.
- The Touch Sensor detects when the button on the front of the sensor is pressed. It can tell when the robot runs into something or whether an object is placed against the robot.
- The Light Sensor measures the brightness of light shining into the front of the sensor. It can distinguish between white, black, and shades of gray, and it is useful both for following lines as well as for measuring the brightness of a light source. The sensor has a small light on the front to measure reflected as well as ambient light.
- The Color Sensor can determine the color of objects. In addition, it can act as a Light Sensor.
- The Sound Sensor measures the level of sound near the robot.

Each NXT motor also contains a built-in Rotation Sensor to measure the distance the motor moves.

LEGO also makes a Temperature Sensor (sold separately), and other companies make sensors for the NXT kits. For example, products from HiTechnic and mindsensors.com include a Compass Sensor, Acceleration Sensor, and Gyroscope Sensor. Vernier makes a wide range of sensors for classroom use and an adapter for using these sensors with the NXT.

The NXT Intelligent Brick (often abbreviated as just "the NXT") is the brains of your robot. The NXT is really a small computer that you program to make your creations move. Instead of a full-size monitor and keyboard, it contains a small display screen and a set of buttons, along with connections for the motors and sensors. When you create a program on your computer using the MINDSTORMS software, you can then download it to the NXT using a USB cable or a Bluetooth connection. When you run the program, the NXT collects data from the sensors and moves the motors, all according to the instructions you provided as the program.

NXT versions

The NXT comes in three main versions: the education set, the original NXT retail kit (released in 2006), and the NXT 2.0 retail kit (released in 2009). Each version has a different mix of building pieces and sensors, as well as different versions of the MINDSTORMS software. All three kits contain the same NXT and motors.

For the purposes of this book, the difference in building pieces is not an issue. The robot we'll be using for the example programs can be built with any of the kits or education sets. Some slight differences will exist (for example, the tires are different in the NXT 2.0 retail kit), but nothing of any real importance.

Table 1-1 lists the sensors included with each NXT kit. The main differences are that the NXT 2.0 retail kit replaces the Light Sensor with a Color Sensor and adds a second Touch Sensor instead of a Sound Sensor.

table 1-1: sensors included with each NXT kit

version	sensors
Education set	2 Touch, 1 Ultrasonic, 1 Sound, 1 Light
NXT retail kit	1 Touch, 1 Ultrasonic, 1 Sound, 1 Light
NXT 2.0 retail kit	2 Touch, 1 Ultrasonic, 1 Color

MINDSTORMS software versions

There are three major versions of the MINDSTORMS software: Version 1.0/1.1, Education 2.0, and Retail 2.0.

NOTE **Most of the material presented in this book applies to all three versions. Even though some new features have been added to the newer versions, the basic process of writing a program has not changed from the original. There are just a few significant differences, such as the change from whole to floating-point numbers, and I'll address those along the way. I tested the example programs with all three versions and will point out any adjustments that you need to make for a particular version.**

version 1.0/1.1

Version 1.0 is the original software released for the NXT. Version 1.1 is an update that includes some speed improvements and bug fixes. There are separate versions for the retail kits and education sets, although only a few differences exist between the two. Each version contains some example robots, and the education set version supports some older RCX sensors (RCX is the pre-NXT LEGO robotics product). Programs are interchangeable between the education sets and retail kits, so I'll refer to both as version 1.1 when pointing out features that are specific to a particular MINDSTORMS software release.

education 2.0

This software version was released in January 2009 by the LEGO Education division. It includes several changes to enhance *datalogging* (the automated collection of data by the NXT), making the NXT more useful for science experiments. The other major change has to do with the way numbers are handled. Whereas version 1.1 uses only whole numbers, the new software uses floating-point numbers that allow digits after the decimal point (for example, 25.123).

retail 2.0

This version of the software is included with the new NXT 2.0 retail kit. Like the Education Software 2.0, this version uses floating-point numbers instead of only whole numbers. It includes some new tools for creating sound and image files for use in your programs, but it doesn't include

the datalogging features from the Education version. This version also includes support for the new Color Sensor.

art and engineering

For me, the most fascinating part of creating a robot is writing the program to make it come alive. This book is all about programming robots using the LEGO MINDSTORMS NXT-G programming environment. You will learn about all the different things NXT can do and how to put all the programming pieces together to create complex robots.

Computer programming is a combination of art and engineering. We tend to call something *engineering* when we have a good understanding of the process involved and can follow a set of logical steps to solve a problem. You can use many engineering principles to make it easier to create your programs: things such as understanding the requirements of your program before you start and performing thorough testing before you consider the program finished. (As you move through this book—and especially with the longer programs toward the end—I'll show you some good programming practices to help you write better programs and some common bad habits to avoid before they cause problems.)

Computer programming has come a long way since the first computer programs. Today we have a good understanding of how to build programs that work well with few errors. However, the basic process of writing a program to solve a particular problem is still more art than engineering. There is no step-by-step process that you can follow. Creating a program usually involves a lot of creativity and ingenuity—things the human mind is good at even though we do not really understand how they work. In my opinion, this use of creative thinking makes programming so much fun. (You may consider your program a work of art. In fact, particularly inventive programs are often described as being *beautiful* or *elegant*.)

As fun as programming can be, it can also be frustrating when things do not work quite the way you want. Mechanical problems can be easier to deal with because you can see how the parts move and what is going wrong, but when a program has a problem, figuring out why can be a bit of a mystery. Throughout this book, I will show you how to find where your program is going wrong and how to fix it. Just remember, solving a mystery should be fun!

qualities of a good program

Many of the decisions you make when creating your programs will depend on your individual taste, and you will develop your own programming style. There is usually more than one correct way to solve a problem. However, a set of three rules is often used to judge the quality of a program. A program should do the following:

1. Perform the desired function
2. Be easy to modify
3. Be understandable by someone who knows the programming language used

The first rule seems pretty obvious, but it is not quite as simple as it may seem. Before you can be sure a program works, you first need to be able to say precisely what it should do. The complete description of what a program should do is called the program's *requirements*. If you are creating a program for a school project or a FIRST LEGO League (FLL) challenge, you might receive the complete requirements before starting. If you're just building a robot for fun, you can make up the requirements as you go along. In either case, you need to know what your robot should do before you decide whether it's a success.

The second rule exists because sometimes requirements change after you start a program. You may find that you can't solve a problem the way you first thought, or you might expand the requirements to solve a harder problem. It is also more likely that a program that is easy to modify will be reused to solve similar problems. Reusing existing programs instead of starting new programs from scratch can save a lot of time.

The third rule is all about keeping things as simple and easy to understand as possible. Programs that are more complex than necessary tend to have more errors and are harder to reuse. In addition to keeping things simple, you can use comments to explain how the program works. Well-placed comments are a simple way to make a program useful to other programmers.

software, firmware, and hardware

Your program is one of three components that work together to control your robot. The program you create is called *software*, which is a set of instructions that a computer can perform. In this case, the computer is the NXT Intelligent Brick. The *soft* part of the word *software* comes from our ability to make changes easily. This is what gives us the power to use the NXT, three motors, and four sensors to create an endless variety of programs.

The program that runs directly on the NXT Intelligent Brick is *firmware*, which is a program that runs on a device (like the NXT) that is not changed often and that is effectively part of the device. Firmware is the program that makes the sound when you turn on the NXT, controls the display, and responds to the buttons on the NXT. When you connect the NXT to your computer with a USB cable or a Bluetooth connection, the MINDSTORMS environment communicates with the NXT's firmware.

NOTE **LEGO may occasionally release updates of the NXT firmware to add new features or fix problems. You can use the MINDSTORMS software to download a firmware update to your NXT Intelligent Brick. You can find firmware updates at *http://mindstorms.lego.com/Support/Updates/*.**

The hardware your program runs on is the NXT. The *hardware* is the physical part of a computer. In addition to the NXT, you can also consider the motors, sensors, and building pieces to be hardware. The hardware does not change; you can put the pieces together in different ways, but what each piece (including the NXT, motors, and sensors) can do doesn't change.

NXT-G

The LEGO MINDSTORMS application is a graphical programming environment for writing programs for an NXT robot. It is called a *programming environment* because it contains all the tools you need to create a program for a robot. This type of application is often called an *integrated development environment* (or *IDE* for short). The MINDSTORMS IDE is considered a *graphical* programming environment because it allows you to create a program by drawing a picture instead of just typing text.

NXT-G is the programming language used by the MINDSTORMS environment. In NXT-G, the individual parts of a program are called *blocks*. There are blocks for controlling the motors, using the sensors, and many other things. You create a program by using the screen to drag blocks around and connect them together. The way your program behaves depends on which blocks you use and how you arrange them.

NXT-G was created as a team effort by LEGO and a company called National Instruments, which makes a wide variety of scientific instruments and the software to control them. One National Instruments product is LabVIEW, which is a graphical programming environment for controlling scientific instruments. NXT-G was created by customizing LabVIEW for use with NXT.

One of my first programming jobs was developing a product for creating control systems used by petroleum refineries. Petroleum engineers used our software to draw all the pipes, pumps, and valves on the screen and connect them. Since then, I have worked on many scientific products that the user can program graphically. Graphical programming is a useful tool for helping people who are new to programming control scientific equipment. Thousands of scientists and engineers all over the world use software tools similar to NXT-G to control all kinds of high-tech equipment (including robots).

When designing NXT-G, the engineers at LEGO and National Instruments faced a difficult challenge. They needed to keep things simple enough for children to use; after all, LEGO is a toy company, and regardless of how much fun MINDSTORMS is for adults, it is still a toy. At the same time, LEGO also wanted to make the system powerful enough to satisfy more experienced LEGO fans.

The result is a remarkable balance between ease of use and programming power, but like all things in life, this balance is not perfect. When you first start out, you may find things that seem odd and difficult to understand (if you have played with data wires, you know what I mean). And after you use NXT-G for a while, you will eventually come across some problems that might have been easier to solve if NXT-G had a few more features.

I've tried to address both of these problems. The first part of this book explains how NXT-G works, because knowing why things behave the way they do helps to avoid some of the confusion that new MINDSTORMS users tend to experience. Later in the book, I show how to get around the limitations you will inevitably run into.

what you'll learn from this book

Knowing some characteristics of a good program is nice, but it's not really enough for you to become successful at writing your own programs. The secret to becoming a successful programmer is knowledge and practice. Throughout this book, I concentrate on the following three areas of knowledge that are important to becoming a successful NXT-G programmer:

- **The behavior of each block.** Learning how each block works is the first step to using it in a program. Although there are many blocks, each of which has many options, learning about each block is not difficult. The NXT-G help file gives a comprehensive description of each block, and it is pretty easy (and fun) to write little test programs that let you discover exactly what each block can do.
- **Joining several blocks together into a working program.** To do this, you need to learn about data wires, variables, and parallel sequences. This is where things get a little more complicated. Learning some details about how NXT-G works can go a long way to evading the confusion that many users experience when they move beyond simple programs. You can avoid a lot of frustration by not only learning how the more advanced features of NXT-G are designed to be used but by also learning the most common problems people encounter and how to debug and fix them.
- **General programming practices.** The three rules listed earlier are the first examples. As we go along, I will introduce more concepts that are useful regardless of the programming language you are using or the type of program you are writing.

Programming is one of those learn-by-doing activities, and this is where practice comes in. Many of the concepts you need to understand will make sense only when you see them in action. The more programs you write, the more comfortable you will become. It's all part of the artistic side of computer programming.

what's next?

In the next chapter, I'll go over the parts of the NXT-G programming environment and then present some easy example programs that show how to use the IDE and some simple programming concepts. The following chapters cover the complete NXT-G language using programs that become more and more complex.

You can download the source code for all the programs in this book from *http://www.nostarch.com/nxt-g.htm*.

2

the NXT-G programming environment

This chapter takes a close look at the NXT-G programming environment and presents a few simple programs. The NXT-G programming environment is fairly complex, with lots of features. This chapter starts with the basics, and later chapters cover some of the more advanced features. All the programs in this chapter use only the NXT Intelligent Brick.

a tour through the MINDSTORMS environment

When you start the MINDSTORMS application, you will see the Getting Started window, shown in Figure 2-1. From this window you can either start a new program or open an existing one.

NOTE **The screenshots in this book are from the NXT 2.0 retail kit version of the software running on Windows XP. They may appear slightly different on your screen if you use a different version or operating system.**

Figure 2-1: The Getting Started window

Click the Go button in the Create New Program section to create a new empty

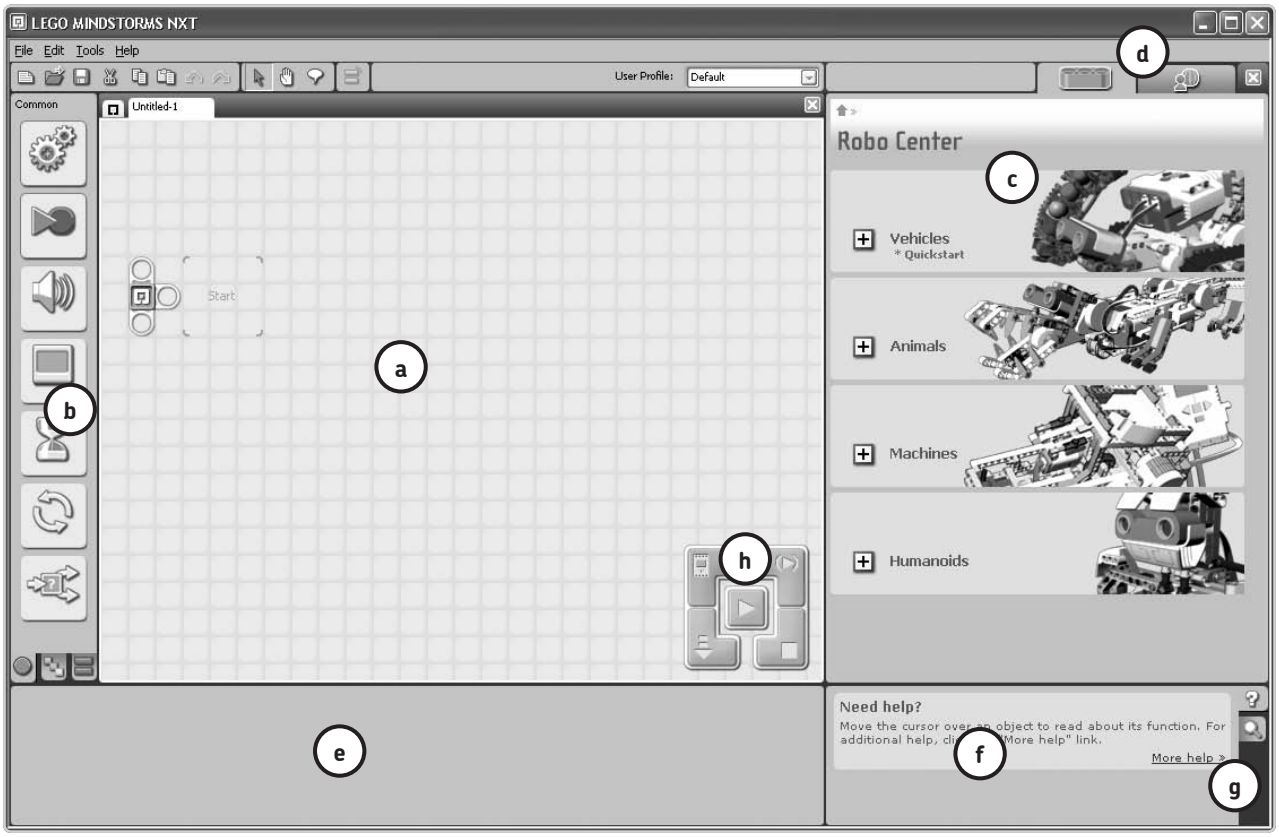


Figure 2-2: The MINDSTORMS NXT environment

and untitled program, as shown in Figure 2-2. Before writing your first program, let's look at the sections of the MINDSTORMS IDE.

a: work area

In the center of the screen is the Work Area, which is also known as the *Program Sheet* because it resembles a sheet of graph paper. Tabs across the top of the Work Area let you select between your open programs. The tab on the left with the little square selects the Getting Started window that is displayed when you first start the IDE.

b: programming palettes

To the left of the Work Area are the Programming Palettes, which contain all the blocks used to create programs. To help keep things simple, there are three palettes, selected using the tabs at the bottom, as shown in Figure 2-3.

Figure 2-3: The Programming Palette tabs

The first tab selects the *Common Palette*. This group has the most commonly used blocks, giving you quick access to the blocks that you'll use most frequently. The center tab selects the *Complete Palette*. All the available blocks, including the ones on the Common Palette, appear on the Complete Palette. The final tab selects the *Custom Palette*. Blocks that you create, called *My Blocks*, will appear on the Custom Palette.

c: robo center

The area to the right of the Work Area can hold either the Robo Center window or the My Portal window (the education set versions use the name *Robot Educator* instead of *Robo Center*). This area provides building and programming instructions for some example projects, with each version of the software providing different projects. I encourage you to work through these projects; they are a great way to become familiar with the MINDSTORMS environment. Closing this area gives you a larger Work Area.

d: my portal window

As mentioned, the area to the right of the Work Area can hold either the Robo Center window or the My Portal window. The

My Portal window shows links to interesting areas on the LEGO MINDSTORMS website. The education set versions of the software connect to the LEGO Education site. When you select the My Portal tab, this window replaces the Robo Center window and should appear something like the following:

e: configuration panel

At the bottom left of the screen is the *Configuration Panel*. Use this area to configure the blocks that make up your program.

f: help panel

At the bottom right of the screen are the Help and Navigation Panels. The *Help Panel* shows a short description of the currently selected block. Click the More Help link to open the full help file, which contains a complete description of every block. Select the Help Panel by clicking the Question Mark tab on the right.

g: navigation panel

When working with large programs, the *Navigation Panel* lets you select which part of your program to display in the Work Area. Select the Navigation Panel by clicking the Magnifying Glass tab. This replaces the Help Panel and displays the entire program, so you cannot see the details of each block, but you can tell which part of the program is displayed in the Work Area. The Navigation Panel for a large program may look like the following:

h: controller

In the bottom-right corner of the Work Area is a group of five buttons called the *Controller*. The Controller is the connection between the programming environment and the NXT Intelligent Brick.

writing an NXT-G program

Writing an NXT-G program is a fairly straightforward process. In the Work Area is what looks like a small white LEGO beam, which is called the *Sequence Beam*. You create an NXT-G program by dragging blocks from the Programming Palettes onto the Sequence Beam. Specify the exact behavior of each block using the Configuration Panel. When you run the program, the NXT executes each block in the order they appear on the Sequence Beam. The blocks are run one at a time, meaning that each block must finish its operation before the next block is started. The program ends when the end of the Sequence Beam is reached. The order of the blocks and the way they are configured determines how the program behaves.

The previous description is actually a little simplistic. A few blocks do not necessarily complete before the next block starts, for example. You can also use more than one Sequence Beam, which complicates how the blocks are run and when the program ends.

NXT-G is a very convenient language for humans, but the NXT needs something a little different. Your NXT-G program is called the *source* or *source code*, and it needs to be translated into a set of instructions that the firmware knows how to execute. The process of translating a program from a human-friendly language to one used directly by the computer is called *compiling* the program.

Once you write a program, use the Controller to compile and download the program to the NXT. *Downloading* is the process of copying the program and any other files the program needs from your computer to the NXT.

your first program

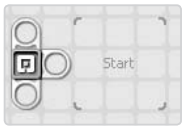
For your first program, use the Sound block to have the NXT say "Hello." When you start the MINDSTORMS IDE, you'll see an area that lets you create a new program or open an existing one. In the Create New Program box, enter **Hello** as the program name, and click the **Go** button, as shown in Figure 2-4. (This box is labeled *Start new program* in the original NXT retail kit and the education set.)

Figure 2-4: Creating a new program

A new Work Area will open. Follow these steps to add a Sound block to the new program:

1. Select the Sound block from the Common Palette, as shown here:

2. Drag the block onto the end of the Sequence Beam, which looks like a little white bar. Place the Sound block right on top of the area conveniently labeled *Start*, as shown here:



Your program should now look like the following:

If you accidentally grab the wrong block or drop it in the wrong place, select Edit ▸ Undo on the menu and start again.

The Configuration Panel for the block should be displayed below the Work Area. If you do not see it, click the block, and the Configuration Panel should appear.

You'll leave most of the items on the Configuration Panel at their default values and change only the Sound File setting. Select **Hello** in the list of available sound files, as shown in Figure 2-5.

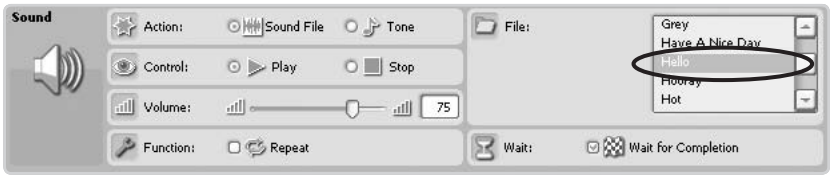


Figure 2-5: The Sound block's Configuration Panel

saving your work

Before continuing, save your program by selecting **File ▸ Save**. When you first save a program, a dialog will let you name the program file and select the location to save it, as shown in Figure 2-6.

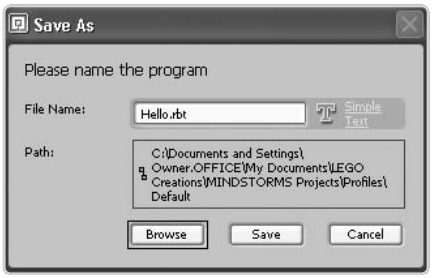


Figure 2-6: Saving your program

The filename defaults to *Hello.rbt* because you entered *Hello* as the program name when creating the new program. If you neglected that step, the name will be something like *Untitled-1*, but you can change it to something more meaningful.

When you click the Save button, the file *Hello.rbt* is created. This file contains all the information about your program, including the blocks you used and how they are configured and arranged. The file format is unique to the MINDSTORMS environment; you won't be able to edit it using other programs.

NOTE **Save your work often. Save before downloading your program and certainly before getting up to answer the phone or walk the dog. Redoing a few hours of work because you neglected to save your program is really annoying.**

MAKING BACKUP COPIES

Most of the time when you are working on a program, you'll make progress. However, every now and then when you change your program, instead of making it better, you end up with a horrible mess, and you can't seem to get back to what you started with. Professional software developers use fancy tools called *source code control systems* to save versions of their work to avoid this problem, but you can get the same benefits by copying working versions of your program to a different folder. Use one of these backup copies if you run into trouble. It's a good idea to save a copy after getting each new feature working and before making large changes. For example, if you are working on a program with four tasks, you might save it as *Task1* after you get the first part working. When you start adding the second task, you might save it as *Task1_Task2*. This way you could always go back a step if necessary.

running your program

Once you save your program, it's time to try it. The first step is to make sure the NXT is turned on and connected to your computer using either a USB cable or a Bluetooth connection. The USB connection is easier to set up (just plug in the cable between the NXT and your computer) but more limiting; a Bluetooth connection can be more difficult to set up but gives you more freedom. If you have problems getting a Bluetooth connection to work, try looking at the NXT hardware forum at *http://www.NXTasy.com/* for several useful tips.

Click the center Play button in the Controller (shown in Figure 2-7) to download and run your program.

When you click the Play button, a dialog should appear with the messages Compiling, Downloading, and Complete. Once the messages finish, your NXT should respond by saying "Hello."

Figure 2-7: The Controller

your second program

The next program, HelloDisplay, is very similar to the Hello program, except that instead of using the Sound block, you'll use the Display block to write *Hello* to the NXT display. One big difference between this program and the first one is that this program won't work the first time. This will give you a chance to look at what to do when your program doesn't work. Use the following steps to create the initial version of the program:

1. Create a new program called *HelloDisplay*. Open the Getting Started window by clicking the small tab on the top of the Work Area that looks like this: 
2. It will be the first tab on the left, before any tabs for programs you have open.
3. Fill in the program name in the Create New Program box, and click the **Go** button as shown earlier in Figure 2-4.
4. Drag the Display block from the Common Palette onto the Sequence Beam. Your program should look like this:

5. The block's Configuration Panel should be displayed, as shown in Figure 2-8. Click the block to select it if the Configuration Panel is not showing.

Figure 2-8: The Display block Configuration Panel

6. To display the word *Hello*, you need to change the Action value from Image to **Text**. Click the arrow beside the word *Image*, and select **Text** from the pop-up box:

7. Changing the Action value causes the Configuration Panel to display a different set of options. It should now look like Figure 2-9.

Figure 2-9: The Configuration Panel after setting the Action value to Text

8. The last step is to replace *Mindstorms NXT* with *Hello* in the Text box. Figure 2-10 shows the completed settings with the changes highlighted.

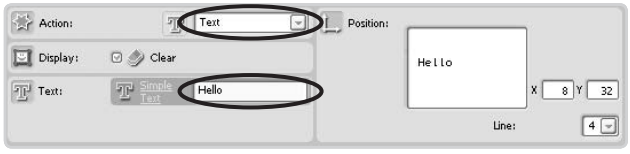


Figure 2-10: Final Display block configuration

Now download and run your program. The NXT will beep twice to let you know it has downloaded a program, but *Hello* will not show up on the display. If you closely watch the display while you run the program, you will see that the NXT starts the program and then immediately says that it's done.

debugging

What happened? To put it simply, this program has a bug. A *bug* is a program error. *Debugging* is the process of finding and fixing errors. Like every programmer, you'll spend a lot of your programming time debugging. Fixing a bug can sometimes be a frustrating process, but it can also be incredibly rewarding to track down and solve a difficult problem. Think of it as solving a puzzle, and remember that you should always have fun!

Of course, some puzzles are easier to solve than others. For example, most people would agree that a Rubik's Cube is a fairly complex puzzle, but even it can be solved by following a simple set of rules (a quick Internet search for *Rubik's Cube Solver* will lead you to a list of solutions).

Like a Rubik's Cube, debugging a program isn't always easy, but it helps to follow a set of rules. Although there is no definitive set of steps for finding a bug, if you follow some simple techniques, you'll find it much easier to debug your programs.

reproduce the bug

First see whether you can reproduce the bug, which just means seeing whether you can get the program to repeat the same behavior. For example, if you try running this program again, it will fail again and will fail the same way every time you run it. This is actually a good thing. A bug that's difficult to reproduce is also more difficult to fix. (Later chapters deal with some trickier ones, but for now we have a reasonably well-behaved bug.)

simplify the program

Normally, the second step is to simplify the program as much as possible, but since this program is already as simple as it can be, there is no way to do so. If it were larger, you could remove pieces to concentrate just on the area with the problem. Another approach is to try to reproduce the same problem using a small test program.

look at the parts of the program

Next, examine the parts of the program to try to figure out what is happening. This program has only two parts—the Display block and the Sequence Beam—so it's likely that one of these is not working quite the way you want.

Think about the difference between how you expect the program to behave and the observed behavior. The program consists of the single Display block, which should write *Hello* to the NXT's display screen, but instead the NXT shows that the program is finished. You know the program ends when all the blocks on the Sequence Beam have been run. A reasonable explanation is that the program ends before you have a chance to read the display because there are no other blocks on the Sequence Beam.

fix the bug

Now that you have a possible explanation for the bug, you can quickly test a possible solution (in most cases, finding the cause of a bug is much more difficult than solving the problem).

You can attempt to fix the program by adding a Wait Time block after the Display block. Use the Wait Time block to tell the program to pause for five seconds before ending, which will give you enough time to read the display.

The Wait Time block does not appear directly on the Common Palette; it is in a group of blocks that wait for different conditions. Hover the mouse cursor over the hour glass icon to make all these blocks appear. Figure 2-11 shows these blocks, with the Wait Time block circled.

Figure 2-11: The Wait Time block on the Common Palette

Use the following steps to fix the program:

1. Drag a Wait Time block onto the Beam to the right of the Display block. Your program should look like Figure 2-12.

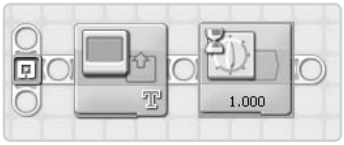


Figure 2-12: The program with the Wait Time block added

2. By default this block pauses for one second. To have a little more time to read the display, change the Until value from 1 to **5**. This will make the program wait five seconds before ending. Figure 2-13 shows the Configuration Panel with the change.

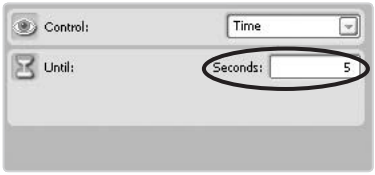


Figure 2-13: Waiting for five seconds

Now when you download and run the program, the display should show *Hello* for five seconds before clearing, which is the behavior you want. Adding the Wait Time block is a successful solution to this bug.

the edit–compile–test cycle

The process you just went through is an example of the typical programming process. In fact, it even has a name: the *edit-compile-test cycle*.

It is very unusual for any program to work the first time. After writing your program, download it to the NXT and test it. Testing will often reveal some problem, so you return to editing. Just keep going through the cycle, adding features and fixing problems. Eventually, all the features are added and all the problems are worked out to give you a complete working program. It can take some patience and perseverance, but it is a great feeling when it finally works.

NOTE **Why didn't the first program have the same problem? It's because the Sound block has a Wait for Completion option that is selected by default (see Figure 2-5). This makes the program wait until the sound plays before continuing. If you uncheck the Wait for Completion option in the first program, it will fail in the same way.**

comments

Programmers use comments to add descriptive text to their programs. Every programming language that I am aware of allows programmers to do this, and NXT-G is no exception. Use comments to tell other programmers how your program works or why you made certain decisions. For example, you could add a comment to the previous program to explain why you added the Wait Time block.

In the previous chapter, I mentioned that a good program should be easy to modify and understandable to other programmers. Good comments are important in achieving both of these goals. It can be very difficult to figure out how a program works just by looking at the settings for each block. A short description in plain English (or whatever language you speak) will make your program much easier to understand. Think of how you might describe your program to a friend. You wouldn't just list the blocks you use; instead,

you'd describe what the program does as a whole, and you might also explain how the more complicated parts of your program work. Use comments to add this type of information to your program so it will be much more useful to other programmers. Comments also help you remember why you wrote a program in a particular way, making it easier to reuse your own programs.

The arrangement and configuration of blocks on the Sequence Beam is all the information that the NXT needs to run your program. Using the previous program as an example, however, a person will want to know *why* you added the Wait Time block. On the other hand, you just need to tell the NXT to wait five seconds. The reason is unimportant. Comments do not affect how a program runs; the NXT will completely ignore them.

adding comments

In this section you will add two comments to the HelloDisplay program. The first comment will describe what the program does, and the second will explain the reason you added the Wait Time block.

the program description

A Configuration Panel exists specifically to hold a description of the program. A reasonable description of this program is *Say "Hello" using the display.* Entering the program description is a simple two-step process:

1. Click the MINDSTORMS icon on the left side of the Sequence Beam (shown here) to open the Configuration Panel for the program description.

2. Enter the description **Say "Hello" using the display** in the box provided in the Configuration Panel, as shown here:

the comment tool

The second comment will be placed before the Wait Time block to explain why that block is there. A reasonable explanation is *Wait 5 seconds before ending the program to give the user time to read the display.*

It may be a little easier to understand the process of adding a comment if you can see the final result. Figure 2-14 shows the program with the comment added. Anyone looking at this program will know why the Wait Time block is there.

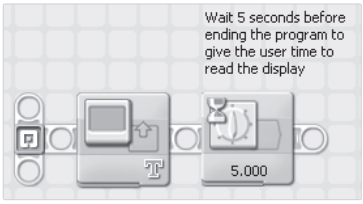


Figure 2-14: Explaining the Wait Time block

Add comments to your program using the Comment tool on the toolbar, as shown here:

Use the following steps for adding this comment:

1. Click the **Comment** tool on the toolbar. When you move your mouse over the program, you should see the I-beam cursor usually used when working with text.

NOTE **The mouse cursor may look a little different if your system settings are not the same as mine.**

2. Click above the Wait Time block. A small black box will appear on the screen to let you know where the comment will be placed, as shown here:

3. Start typing the comment **Wait 5 seconds before ending the program to give the user time to read the display**. Press the ENTER key to move to the next line when the text gets wider than the block (it's easier to read if the comment does not extend way past the block).

4. When you finish adding a comment, you can start another one by clicking in the Work Area. When you finish adding comments, select the Pointer tool from the toolbar.

5. This tells the IDE that you want to go back to selecting blocks. When you move your mouse cursor over the program, you should now see the normal pointer, as shown here:

rules for working with comments

The following are a few things you should know about comments:

* Pressing ENTER while typing a comment makes it go to the next line.
* Clicking the text of a comment lets you change the comment.
* Double-clicking in the Work Area starts a comment.
* Clicking the edge of a comment selects the comment.
* You can delete the selected comment by pressing the DELETE key.
* You can move a selected comment by dragging it with the mouse; just make sure to click the edge of the comment and not the text.

the configuration panel

Before moving on to the next program, let's take a closer look at the Configuration Panel. When you select a block in your program, the Configuration Panel for that block displays in the lower-left corner of the IDE. This is where you set the options that control exactly how the block behaves. You will spend a lot of time working with the Configuration Panel when writing an NXT-G program.

Each block has its own Configuration Panel, and subsequent chapters discuss the details of each block. In this section, I concentrate on how the Configuration Panel works in general. The National Instruments engineers did a good job of using a consistent look and feel across the Configuration Panels for all the blocks, which is one of the features that make NXT-G easy to use.

I'll use the Configuration Panel for the Sound block as an example for this discussion. This is a fairly typical block, and all the ideas discussed here are repeated in many other blocks.

general layout

Each block has a different set of items you can configure, but those items tend to be arranged in the same way. Figure 2-15 shows the Configuration Panel for the Sound block. This is how the Configuration Panel looks when you first add the Sound block to a program, before you make any changes.

Figure 2-15: The Configuration Panel for the Sound block

The items are arranged in two columns. The items you set first will be in the left column, with the most important ones near the top. For example, you can use the Sound block to play either a sound file or a single musical tone. This is controlled by the Action item, located at the top of the first column. The Action is listed first because the type of sound affects how some other items behave.

changing panels

The Sound block has many options, and some of the choices can only be used together. In the Hello program, you left the Action item set to Sound File and selected Hello from the Files list on the right. However, it only makes sense to select a sound file from the list when you select Sound File for the Action. When you select Tone, you need to make different choices. Figure 2-16 shows the Configuration Panel with the Tone action selected. Compare this with Figure 2-15. Notice that the right side of the panel has changed. The list of files has been replaced with items used to select a note to play.

Figure 2-16: Configuration Panel for playing a single tone

Many blocks have Configuration Panels that change like this, where an item on the left side controls the choices that appear on the right. This allows a single block to have several different uses without displaying a confusing list of conflicting options.

disabled items

Occasionally, you may find that the option you want to select is disabled. A disabled item appears on the Configuration Panel but is light gray and you cannot select it. This happens when some other incompatible choice is selected. For example, the Sound block can play a sound or stop a sound that is currently playing. When you select Stop for the Control item, the rest of the choices become disabled, as shown in Figure 2-17. This makes sense because there is no point in setting the volume or selecting a note to play when you want the sound to stop.

Figure 2-17: With Stop selected, the rest of the choices are disabled.

a block's configuration icons

You can select a block and examine its Configuration Panel to see exactly what it does. However, you can look at the Configuration Panel for only one block at a time. Fortunately, the way a block is displayed in the program changes based on some of the most important configuration items. Here is the way the Sound block looks in the Hello program:

Notice the three icons along the bottom. These icons change based on your configuration choices. The icon on the left tells you that the action is set to Sound File. The same icon appears next to the Sound File choice on the Configuration Panel. The middle icon shows that you are playing a sound, and the third icon shows the Volume setting.

This is how the Sound block appears when you select Tone for the Action setting:

The icon on the left has changed to a small musical note to match the Action setting.

When the Control item is set to Stop, the icons for the Action and Volume settings are not shown; only the Stop icon appears:

Each block uses a different set of icons because the blocks all have different settings. If you are not sure what an icon means, look at the Configuration Panel for the block. The icons shown on the block will also appear next to one of the settings.

conclusion

This concludes our tour of the MINDSTORMS environment. I covered the basics that you will need to create simple NXT-G programs, and you'll get plenty of practice as we progress. I will introduce more advance features as we need them.

The next step is to build a simple TriBot to use with the example programs in the following chapters. Then I'll introduce the many blocks available in NXT-G and show you how to put them together to make the TriBot perform a variety of tasks.

3

the test robot

You'll need a robot to use as a test bed to run the example programs in the rest of this book, and you'll build that robot in this chapter. You'll construct a simple, three-wheeled TriBot that's easy to build with NXT parts. The robot will incorporate all the NXT sensors, and you'll be able to use it as a single general-purpose robot in testing a wide variety of programs.

Figure 3-1 shows the finished TriBot. The robot on the left was built using the NXT 2.0 retail kit, and the one on the right was built using the education set.

Figure 3-1: TriBots built from the NXT 2.0 retail kit and the education set

You can build this model with any NXT kit, though the building instructions here use the parts from the NXT 2.0 retail kit. If you build the robot using the original NXT retail kit or the education set, note that just a few differences matter for this model.

The NXT 2.0 retail kit includes flat tires, and the other kits include the larger balloon tires (see Figure 3-2).

Figure 3-2: Flat and balloon tires

The parts used for the caster wheel (the third wheel at the back of the TriBot) also differ between the NXT 2.0 retail kit and the earlier kits. Figure 3-3 shows the parts used for the caster wheel. If you use the NXT 2.0 retail kit, use the 20-tooth gear shown on the left. If you use the original NXT retail kit or the education set, use the two belt wheels shown on the right.

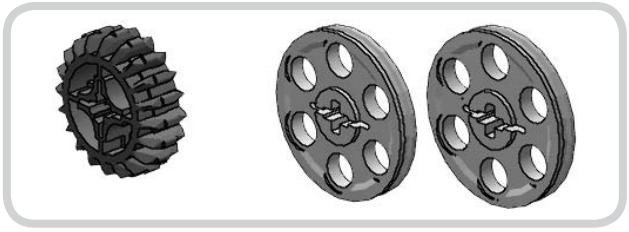


Figure 3-3: Caster wheel parts

If you use the education set, you'll need to make a couple of substitutions. Referencing Figure 3-4, use the beam on the left (with two holes on the short side) for any version of the retail kit and the one on the right (with three holes on the short side) for the education set.

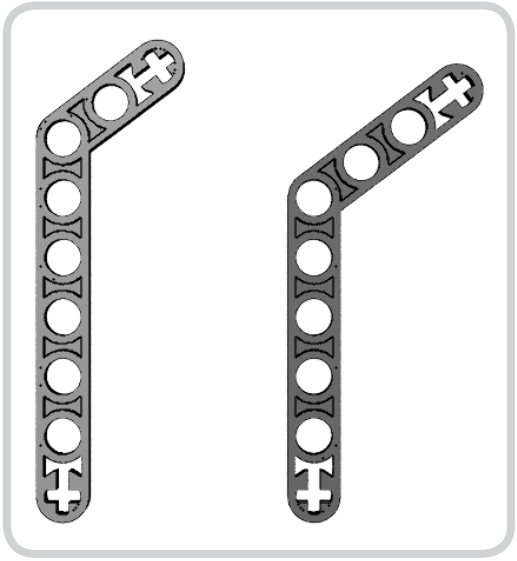
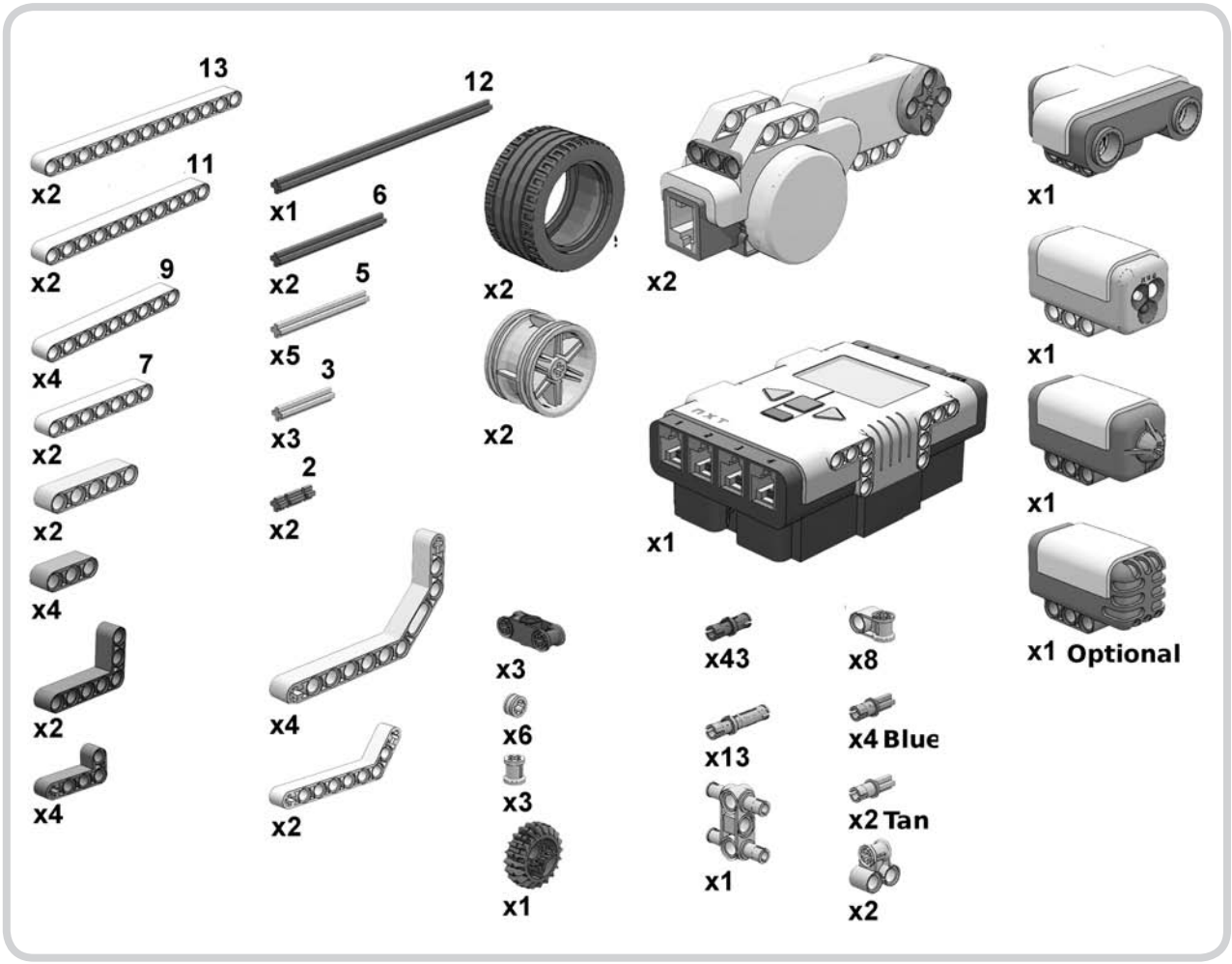


Figure 3-4: Part substitution for the education set

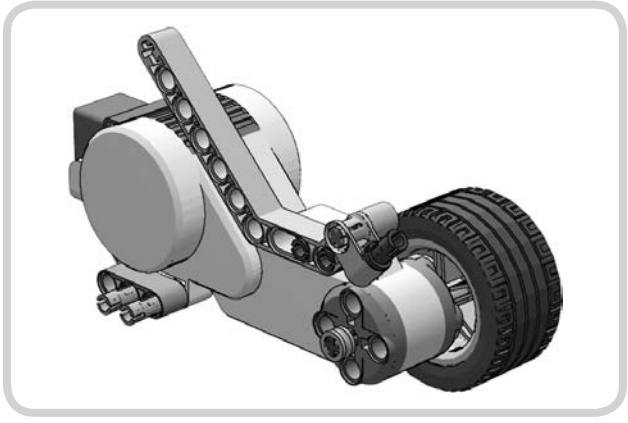
NOTE **Another difference between the three kits is the color of some parts. For example, some beams are white in the retail kits and gray in the education set, and some parts that are gray in the older kits are colored in the new kit. Just make sure the parts are the correct size and shape; it doesn't matter if the color doesn't match the description here.**

The following shows the parts you need if you use the NXT 2.0 retail kit. If you use one of the other kits, make the substitutions noted previously.

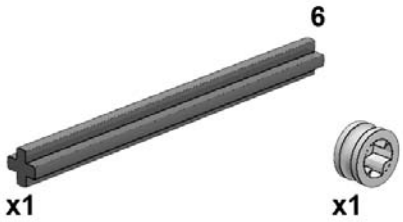


right-side motor

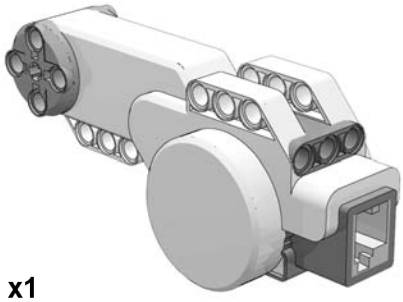
Begin by building the motor and wheel assembly for the right side of the TriBot, as pictured.

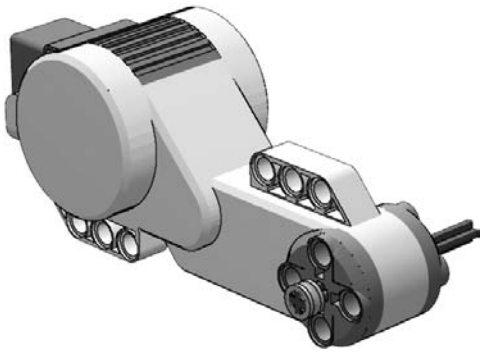


1

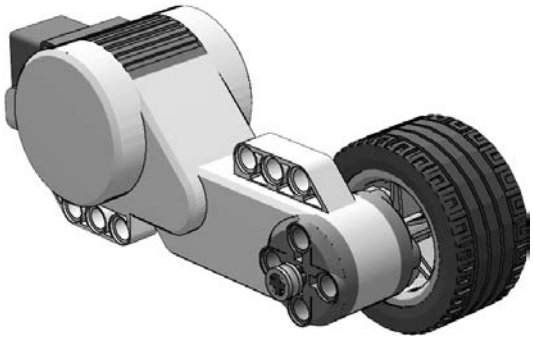


2



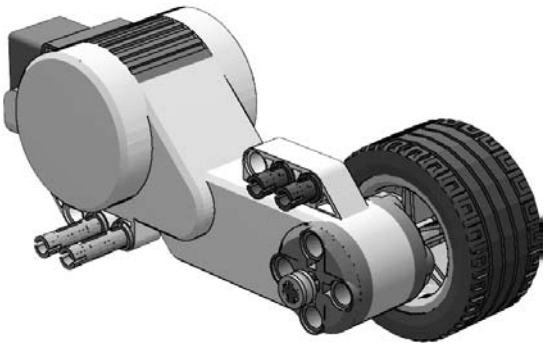


3

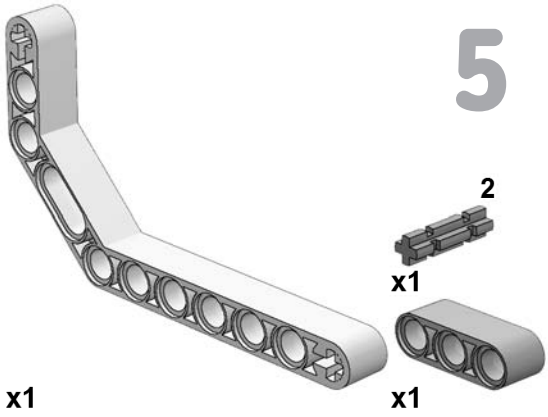


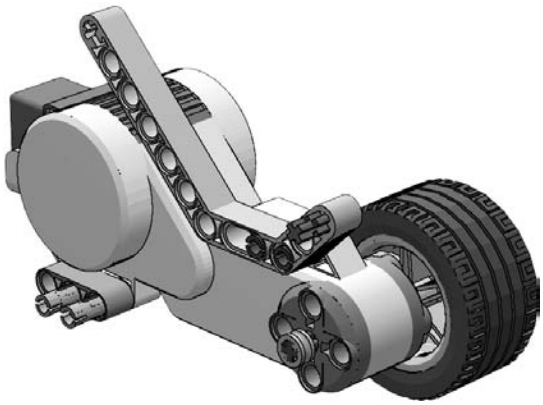
4

The long pin is blue in the NXT 2.0 retail kit and black in the earlier kits.

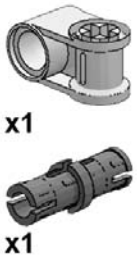


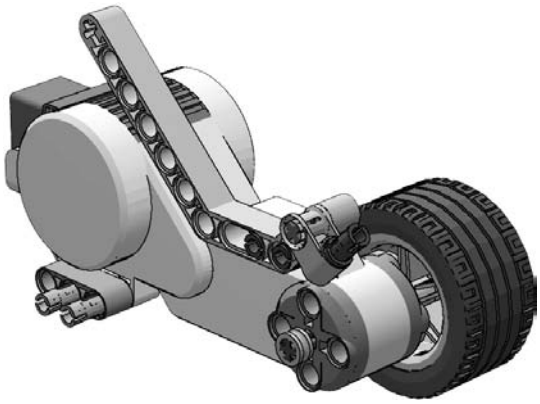
5





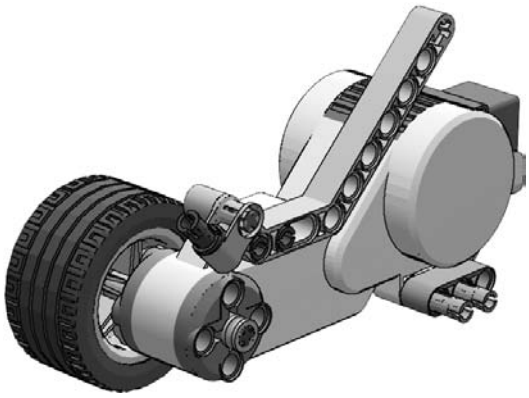
6



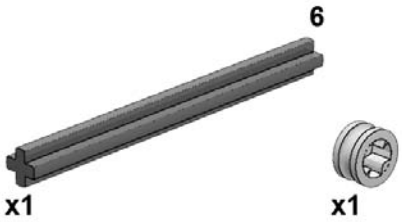


left-side motor

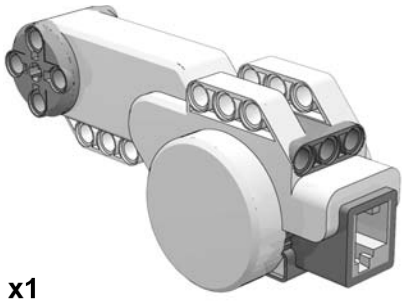
Follow the pictured steps to build the motor for the left side.

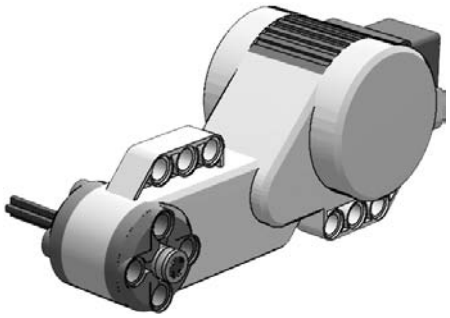


1

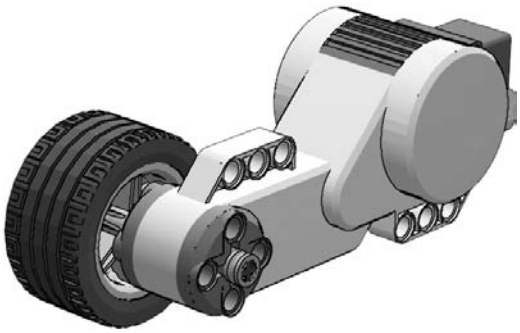


2

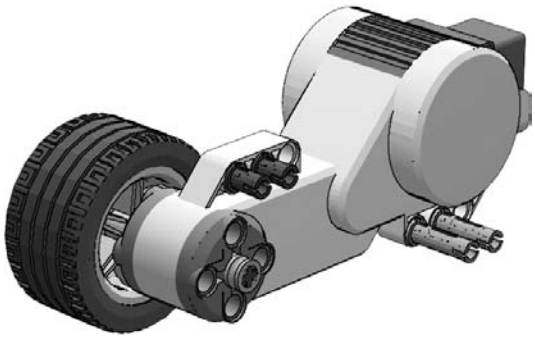




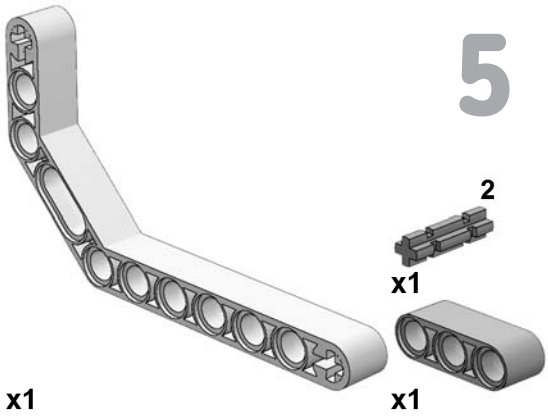
3

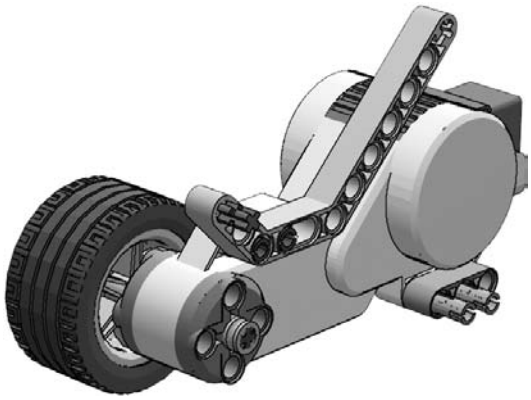


4

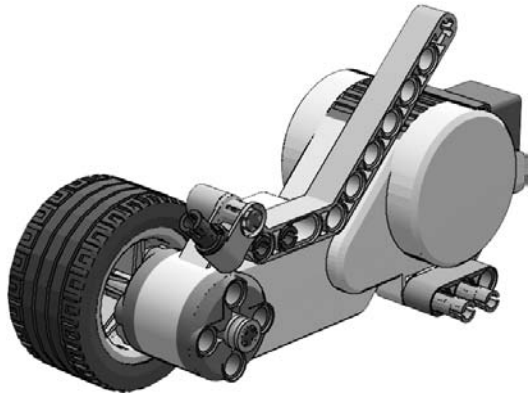


5



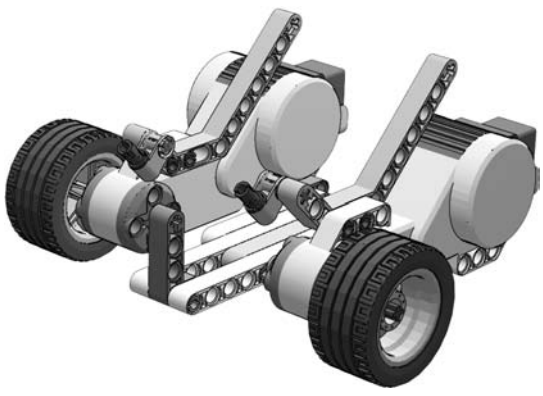


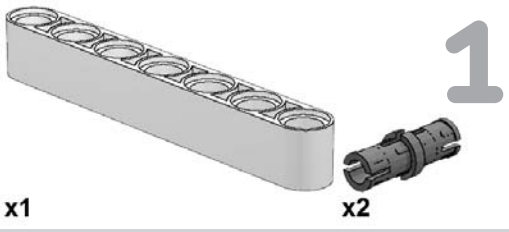
6

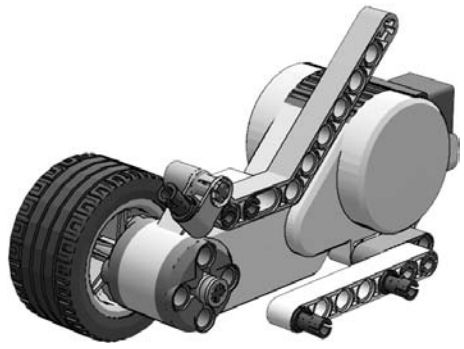


chassis

Now build the chassis that connects the two motors.

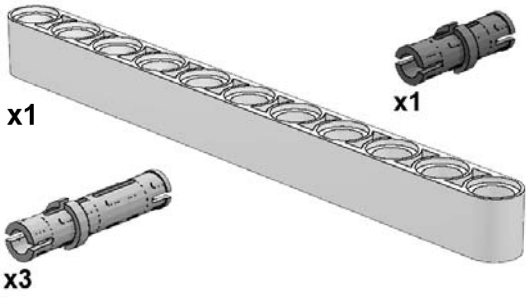


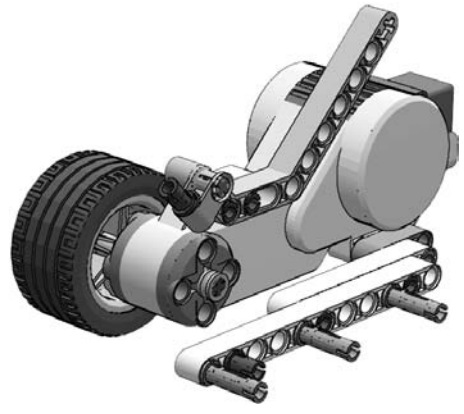


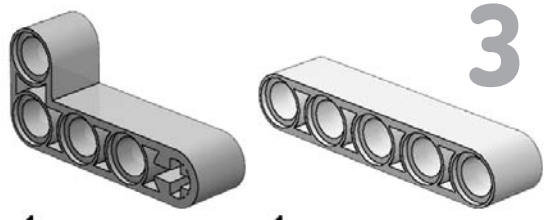


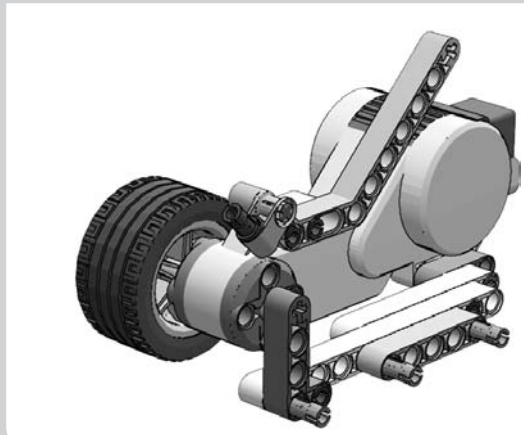
2

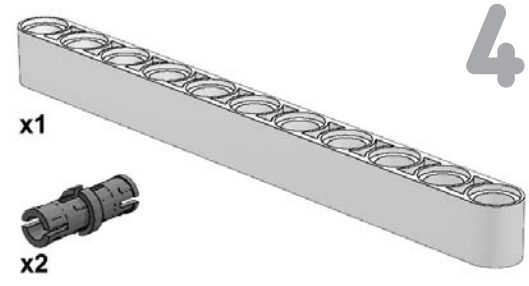
The 11-hole beam added in this step should be connected to the 7-hole beam added in the previous step so that only the last hole of the 7-hole beam is left showing.

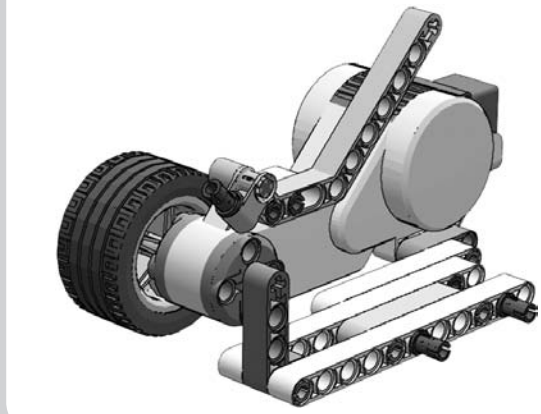


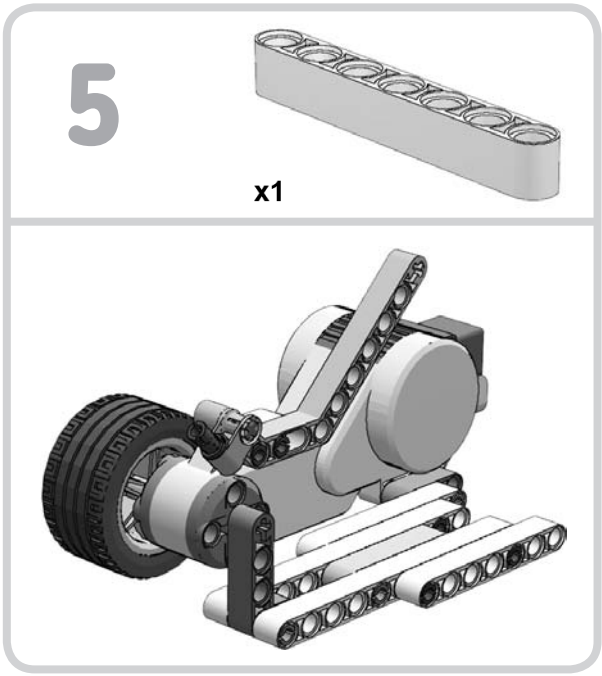




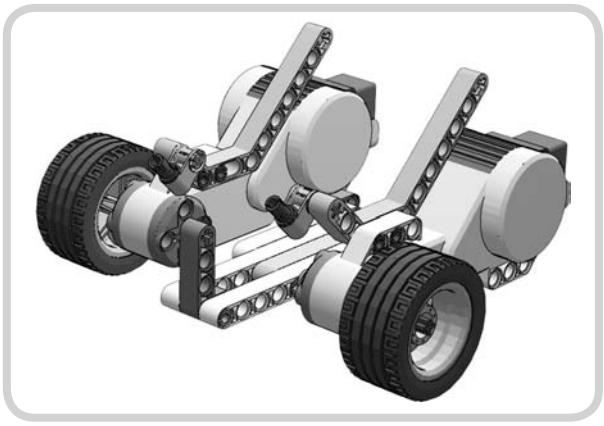




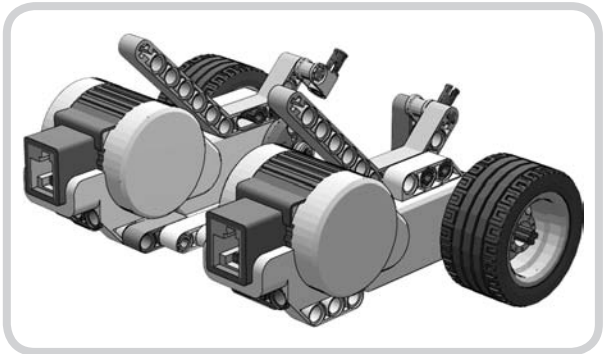




Now attach the right motor assembly.



See the picture for the view from the back.



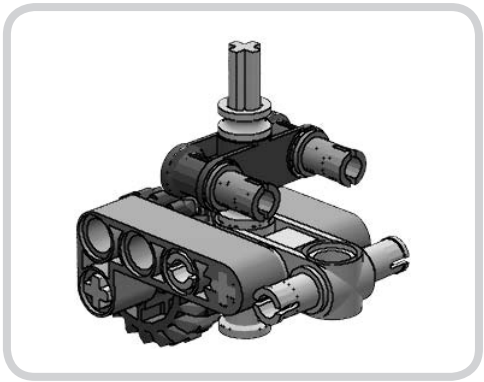
caster wheel

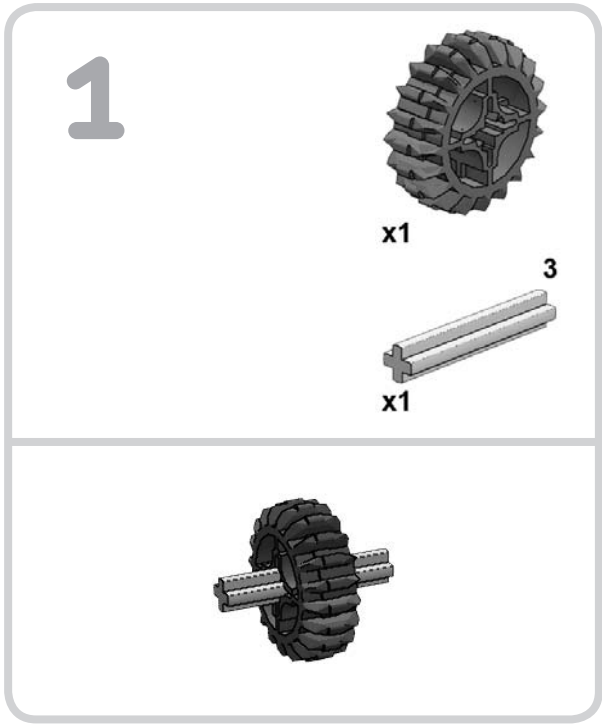
In this section, you will attach the caster wheel to the back of the TriBot. I've included two sets of building instructions because of the differences in the height of the tires and the parts in each kit. The first set of instructions is for the NXT 2.0 retail kit. If you're using the original NXT retail kit or the education set, use the instructions in the following section.

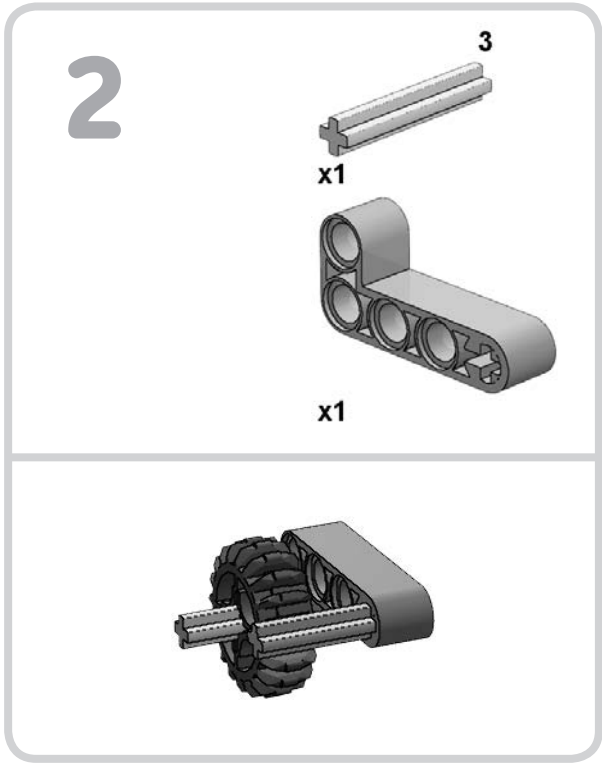
If you're not sure which kit you have, compare your tires with the ones shown in Figure 3-5. If you have the tire on the left, then you have the NXT 2.0 retail kit and should follow the directions in "Caster Wheel for the NXT 2.0 Retail Kit" below. If you have the balloon tire on the right, then you have either the original NXT retail kit or the education set and should skip to "Caster Wheel for the Original NXT Retail Kit and Education Set" on page 27.

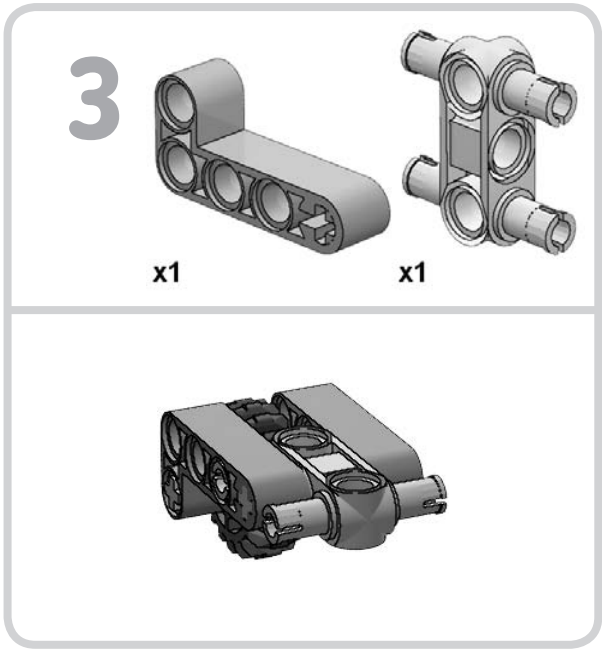
Figure 3-5: Flat and balloon tires

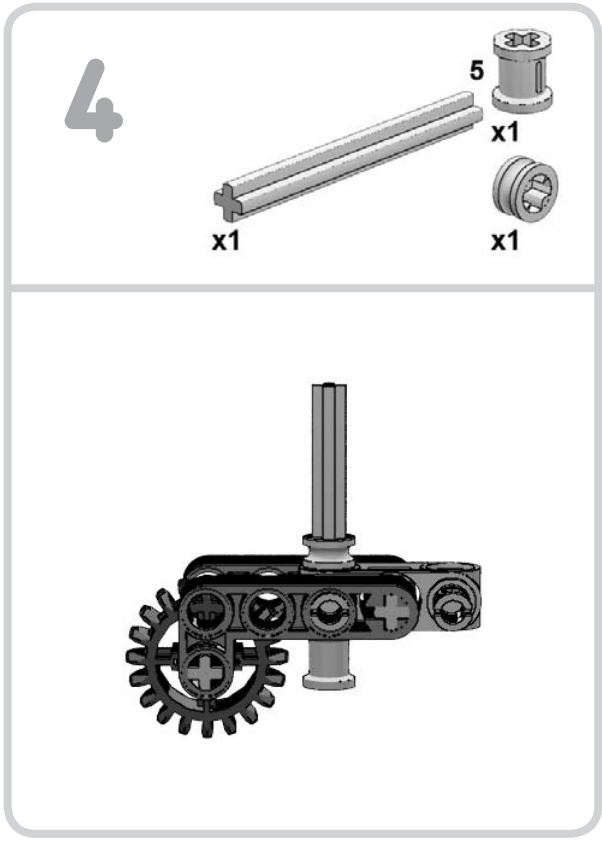
caster wheel for the NXT 2.0 retail kit



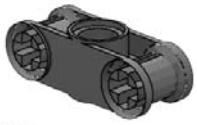

1
x1
3
x1


2
3
x1
x1

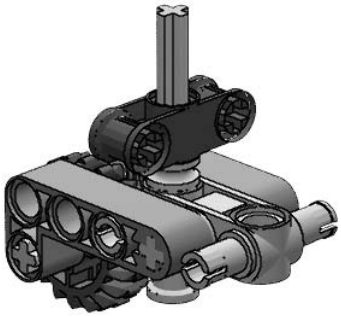

3
x1
x1


4
5
x1
x1
x1

5

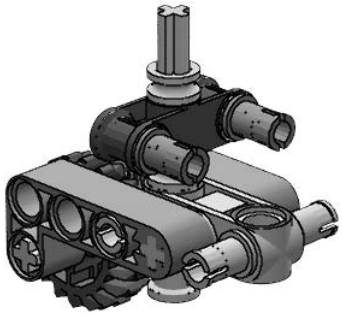


x1



6

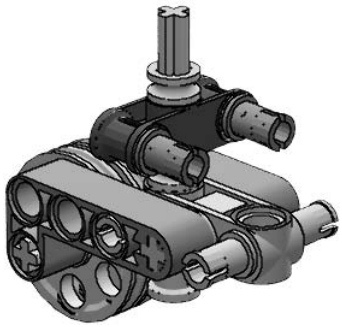
x2 x1



If you have built the caster wheel with the NXT 2.0 kit, proceed to "Attach the Caster Wheel" on page 29.

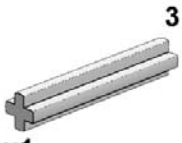
caster wheel for the original NXT retail kit and education set

Follow these instructions if you use the original NXT retail kit or the education set.



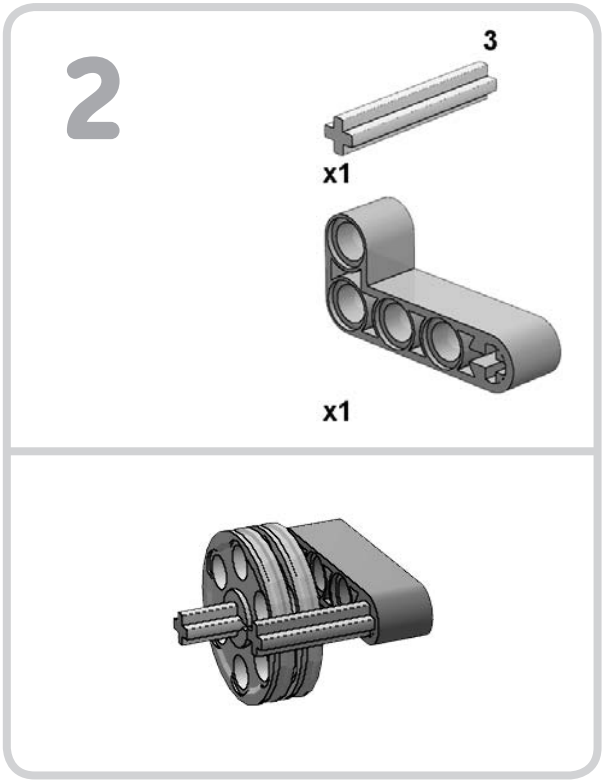
1

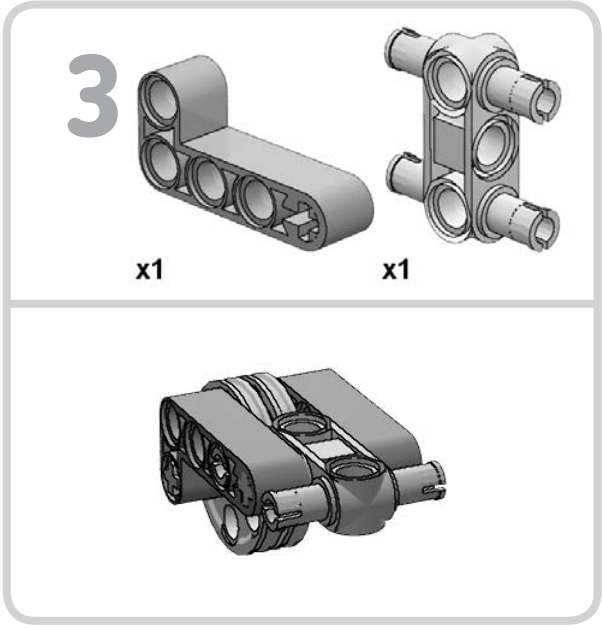
x2

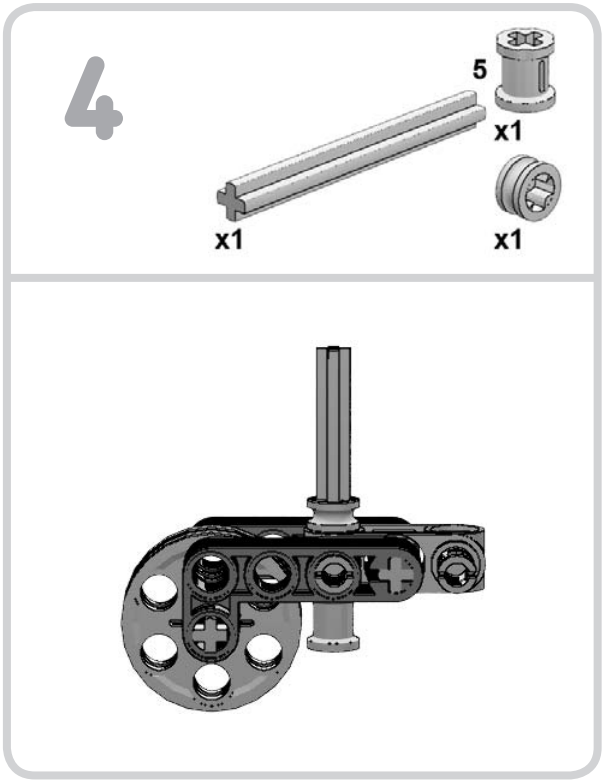


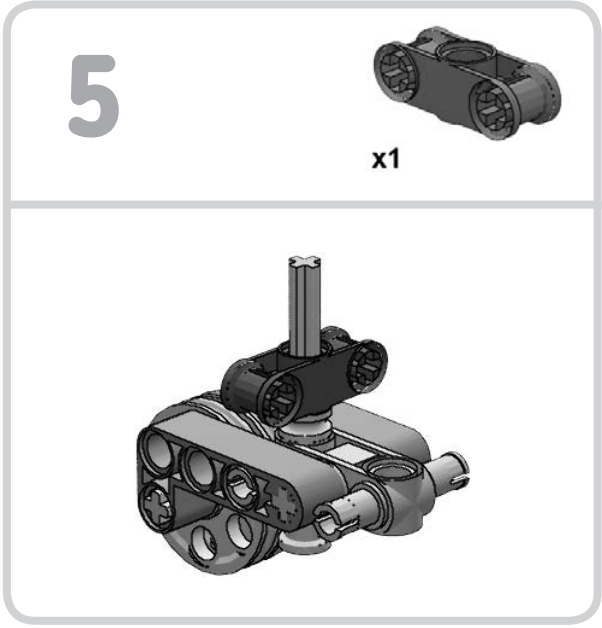
3

x1

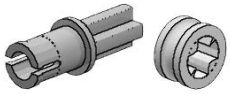

2
3
x1
x1

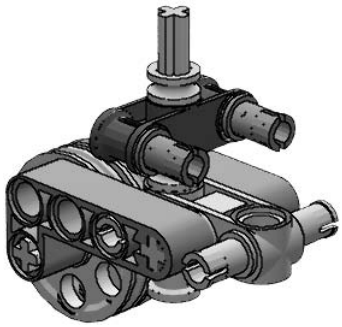

3
x1
x1


4
5
x1
x1
x1


5
x1

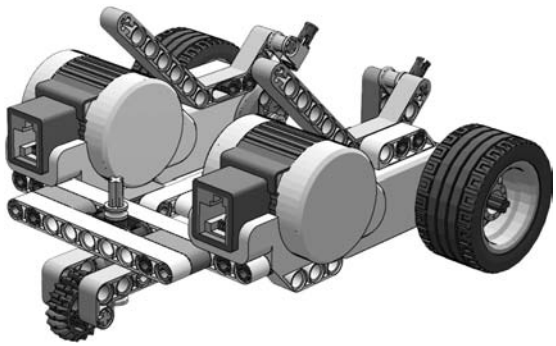
6





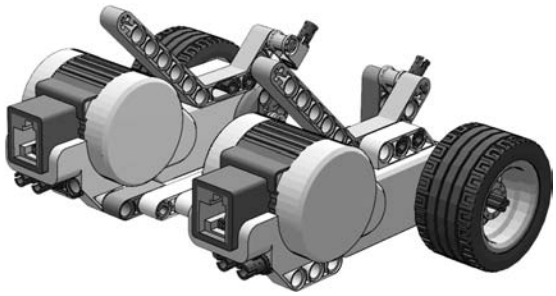
attach the caster wheel

Next attach the caster wheel to the back of the robot.

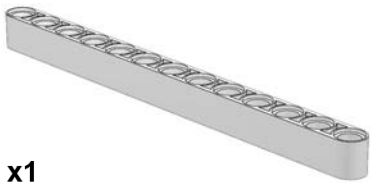


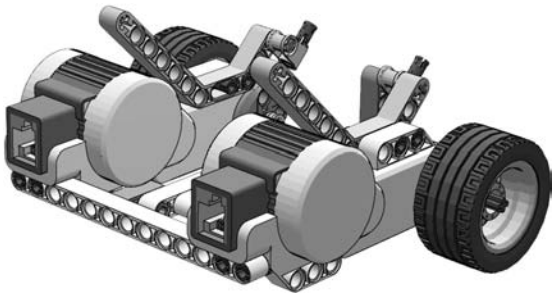
The next step adds two pins to the back of each motor. On each side, the pins should be in the two outside holes, leaving the inside hole (toward the center of the TriBot) empty.

1

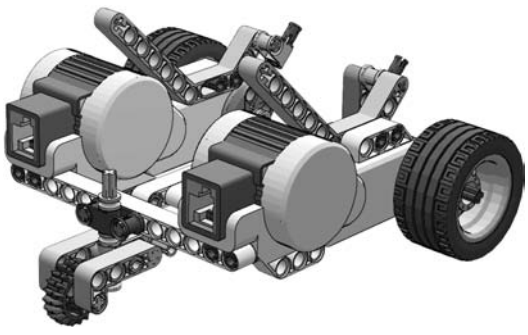


2





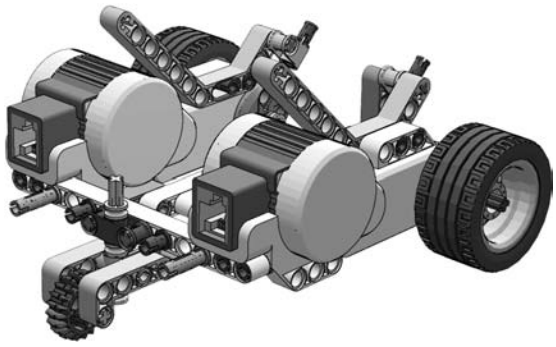
Now attach the caster wheel to the center of the beam.



3

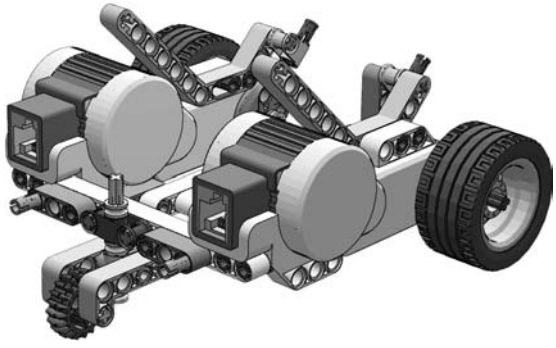
x2

x2



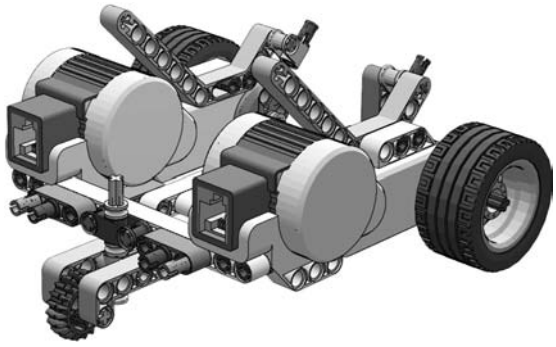
4

x2

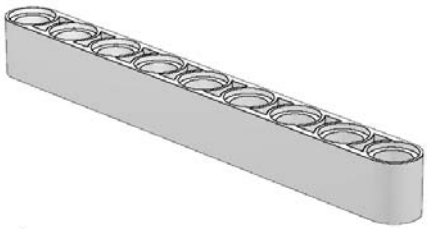


5

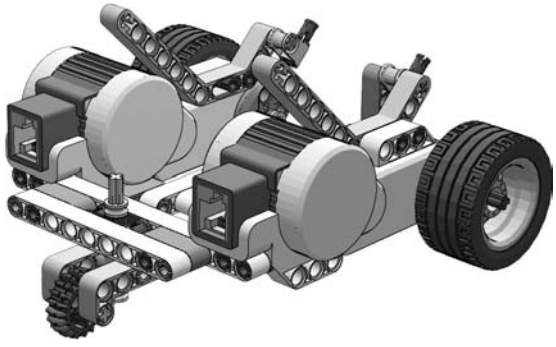
x2



6

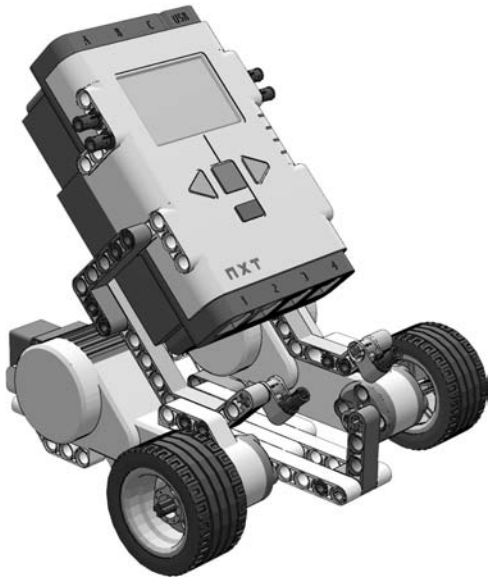


x1



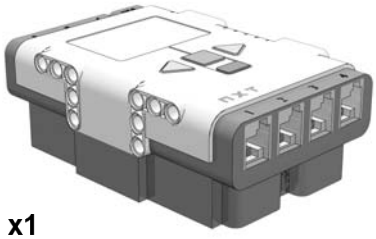
add the NXT

In this step, you'll add the pins you need to the NXT and then attach the NXT to the motors.

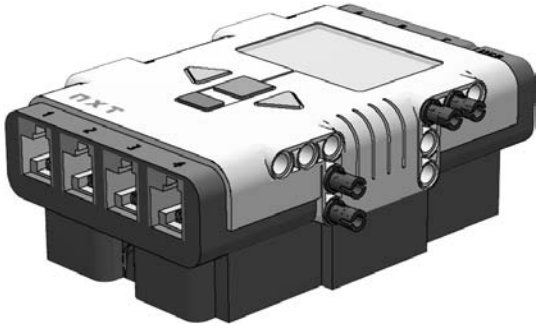


1

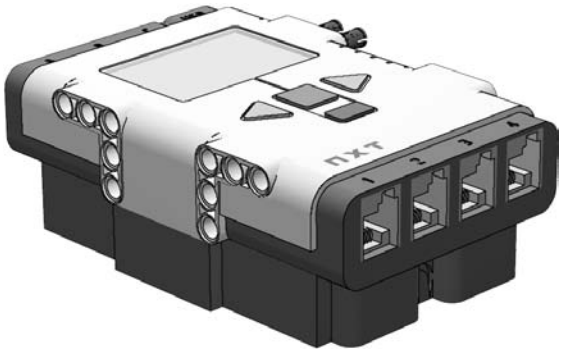
x4



x1

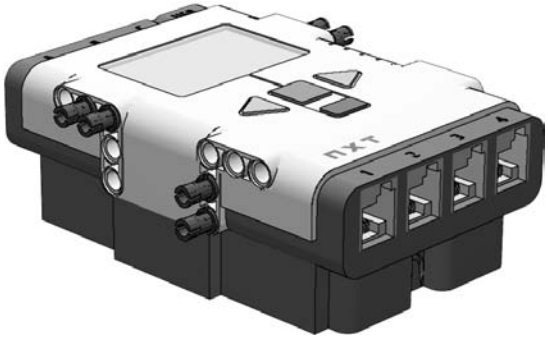


Now turn the NXT around to add pins to the other side.



2

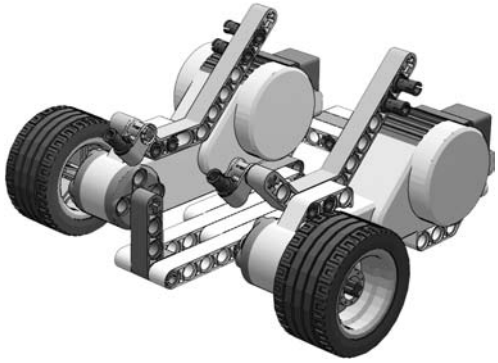
x4



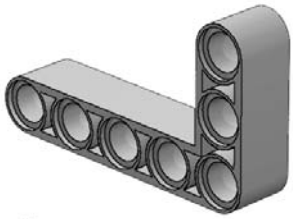
Now attach the NXT to the motor assembly.

3

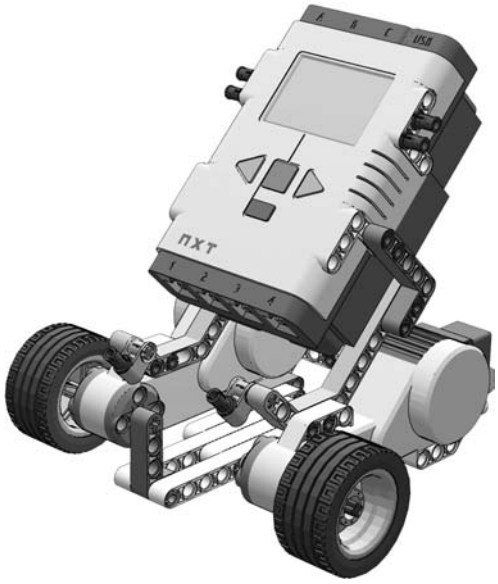
x4



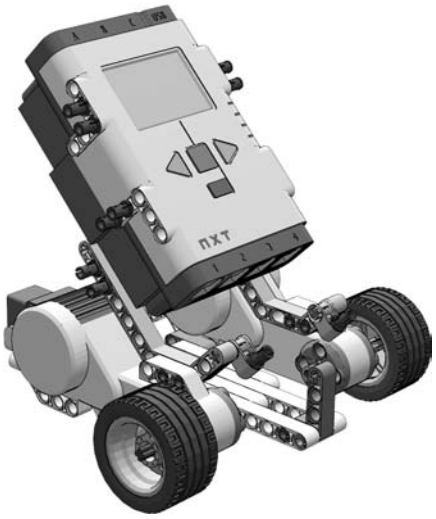
4



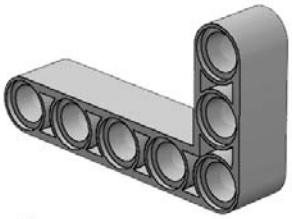
x1



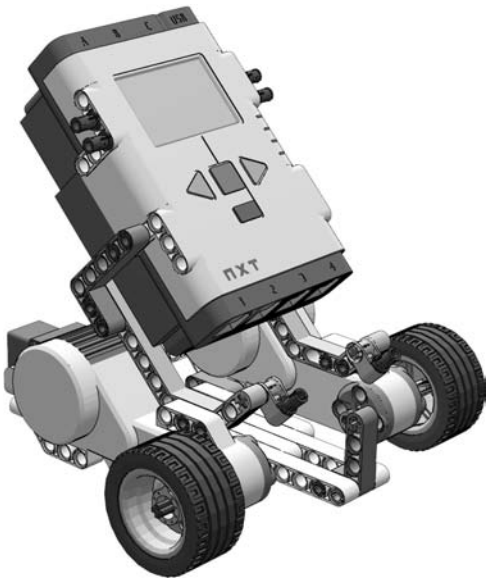
Now flip the robot around to attach the NXT on the other side.



5

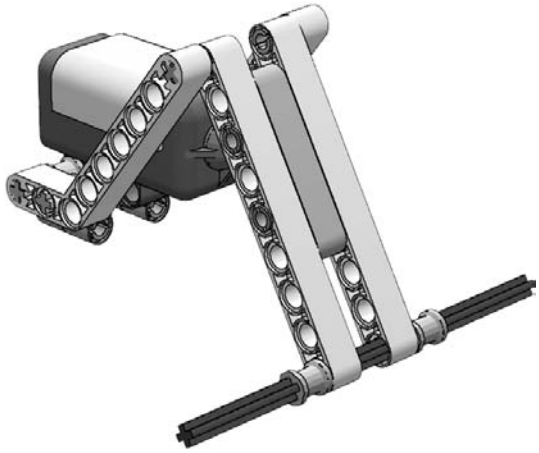


x1

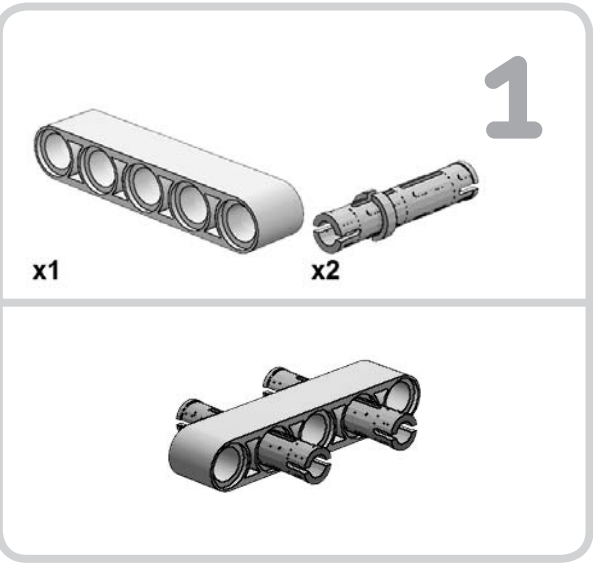


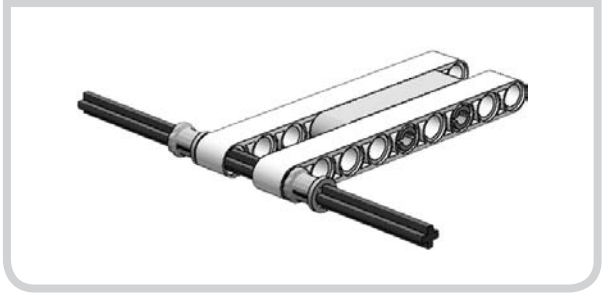
touch sensor bumper

In the following set of steps, you'll build a bumper using the Touch Sensor.

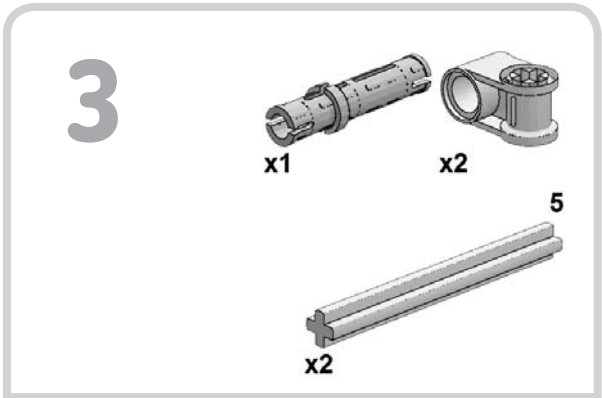


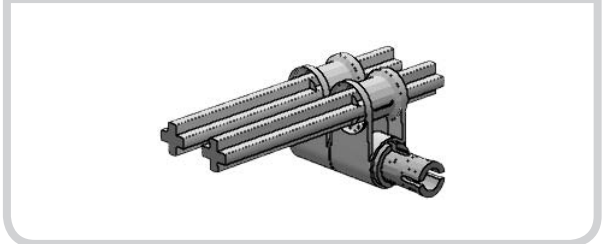
Start by building the bumper arm.

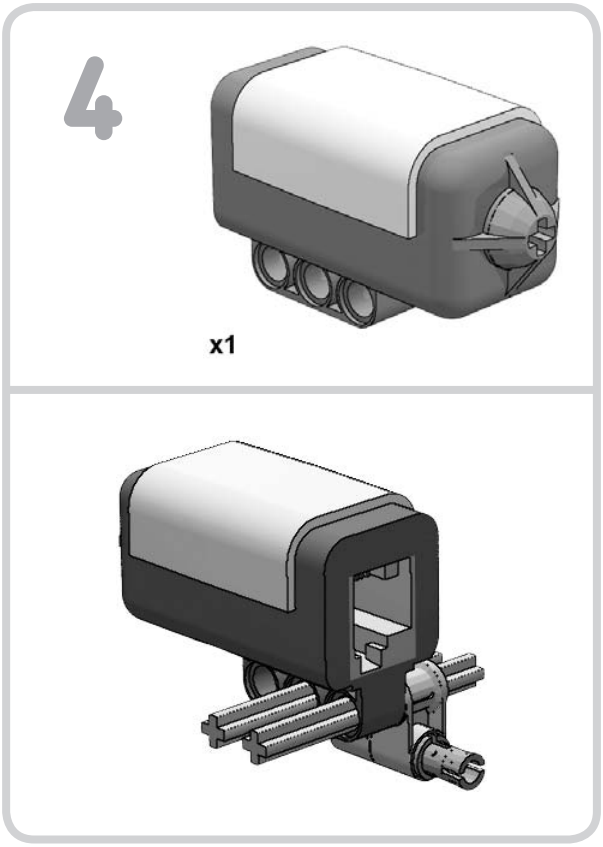


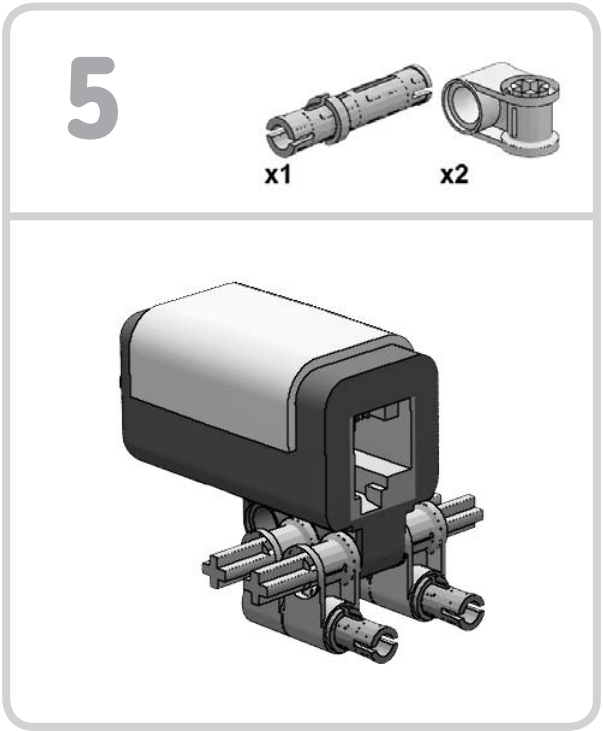


Now build the rest of the bumper.

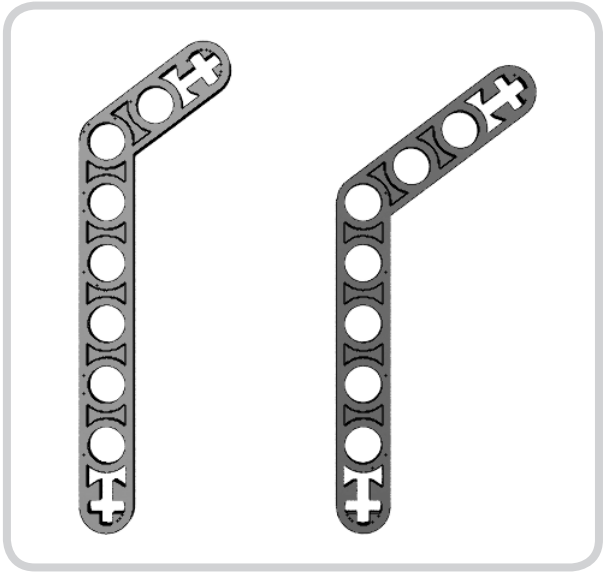


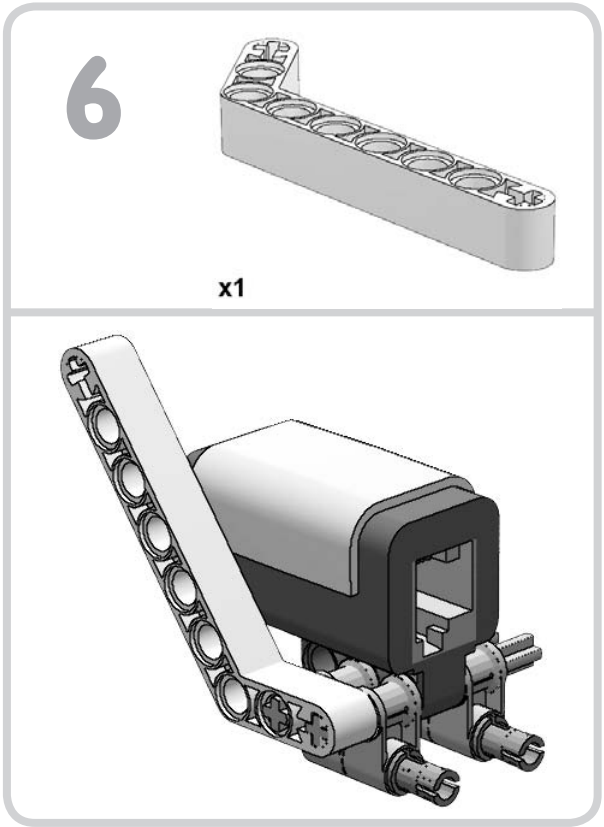




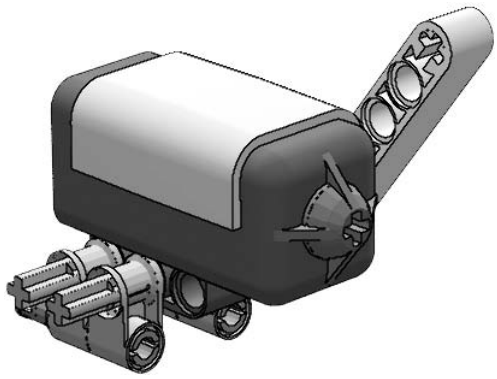


If you are using the education set, you need to make a substitution for the angled beam used to hold the bumper arm. The following instructions use the beam shown here on the left. The education set doesn't contain this piece, so use the beam pictured on the right instead.





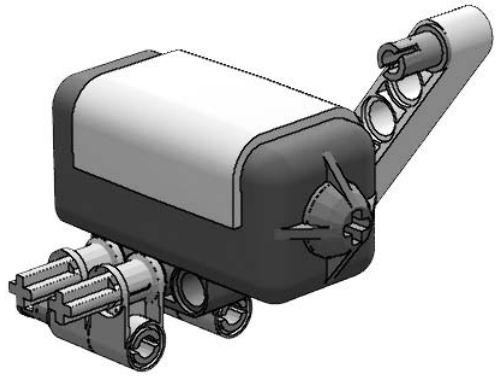
Turn the sensor around for the next set of steps.



Use the tan pins (not the blue ones) to attach the bumper arm to the sensor. The tan pins have less friction, so the arm swings more easily.

7

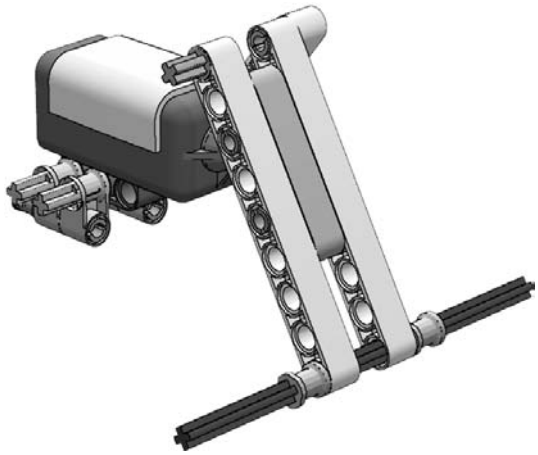
x1



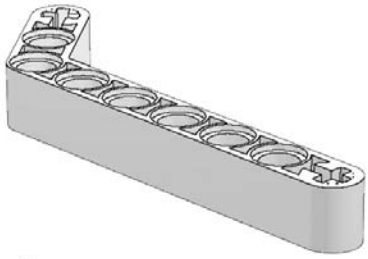
Now add the bumper arm and a pin to connect the arm to the beam you will add in the next step.

8

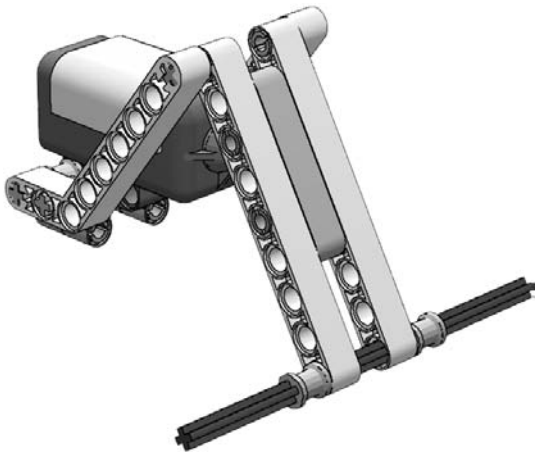
x1



9



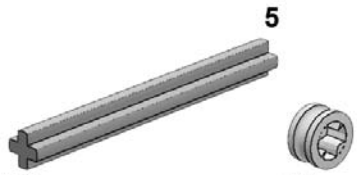
x1

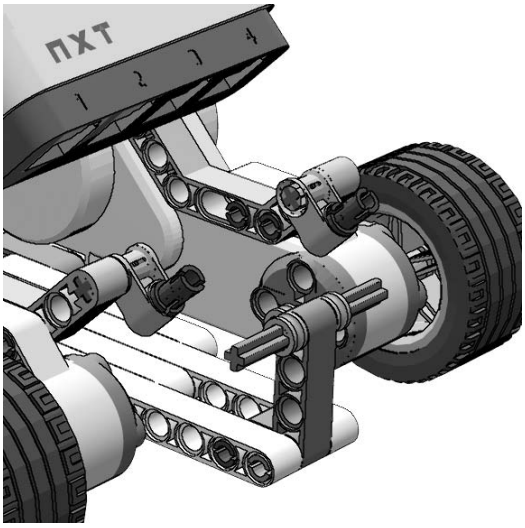


attach the bumper to the chassis

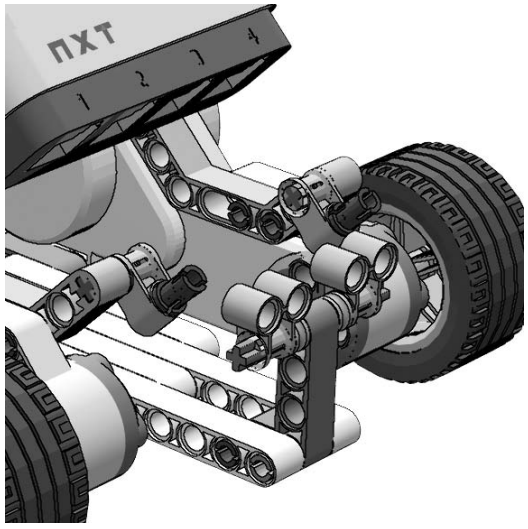
To attach the bumper, first you need to add some connectors to the post at the front of the chassis.

1





2

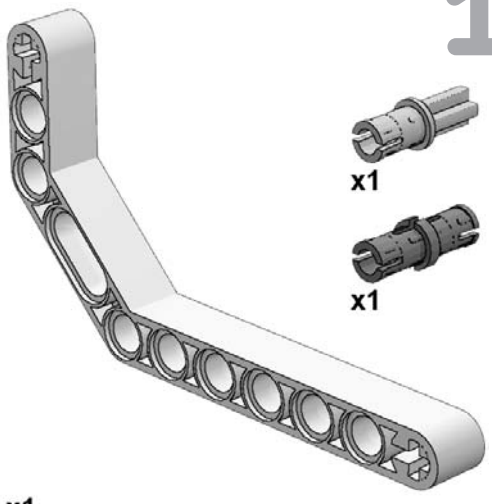


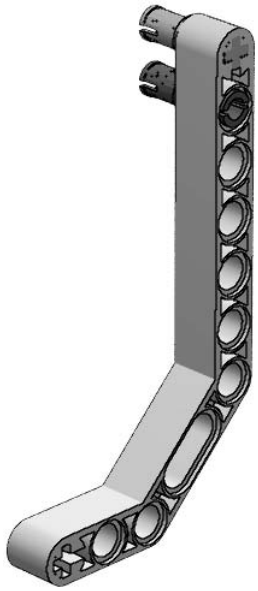
Now add the bumper assembly. The pins at the back of the bumper fit into the two inner pin holders (the ones closest to the post).

ultrasonic sensor

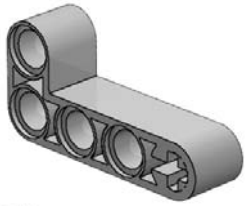
The next step is to mount the Ultrasonic Sensor so it points forward. First build an arm to hold the sensor, and then attach it to the NXT.

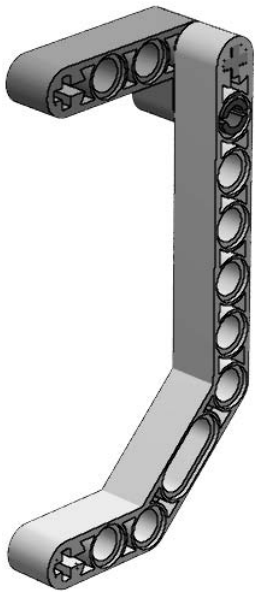
1



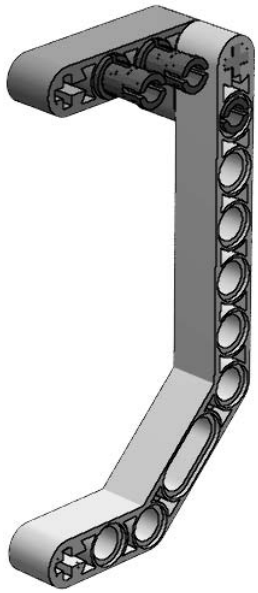


2

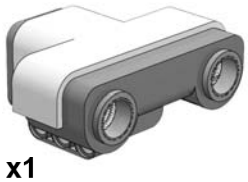




3



4



x1

Now attach the arm to the side of the robot.

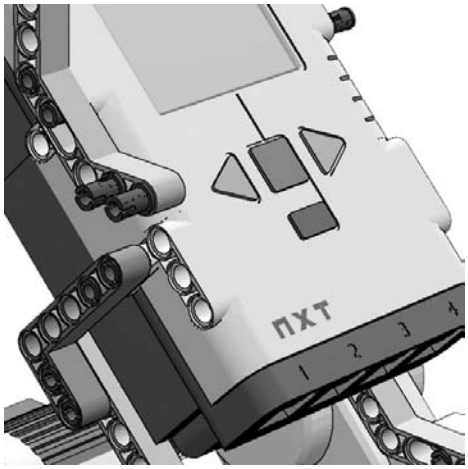
sound sensor

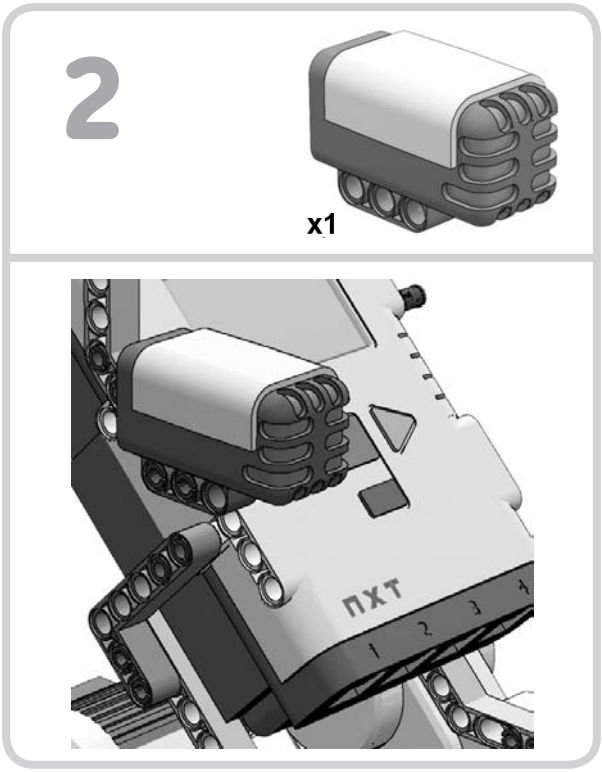
The Sound Sensor attaches to the bottom of the arm. The NXT 2.0 retail kit doesn't come with a Sound Sensor, so you can skip these steps if you have that kit.

1

x1

x1

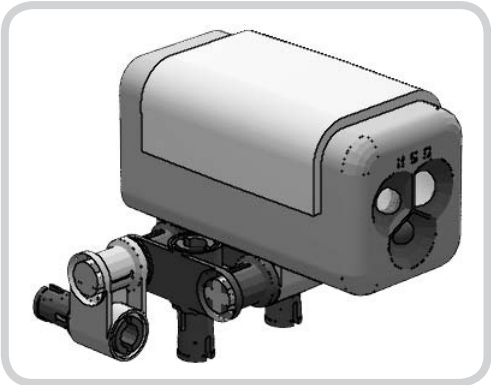


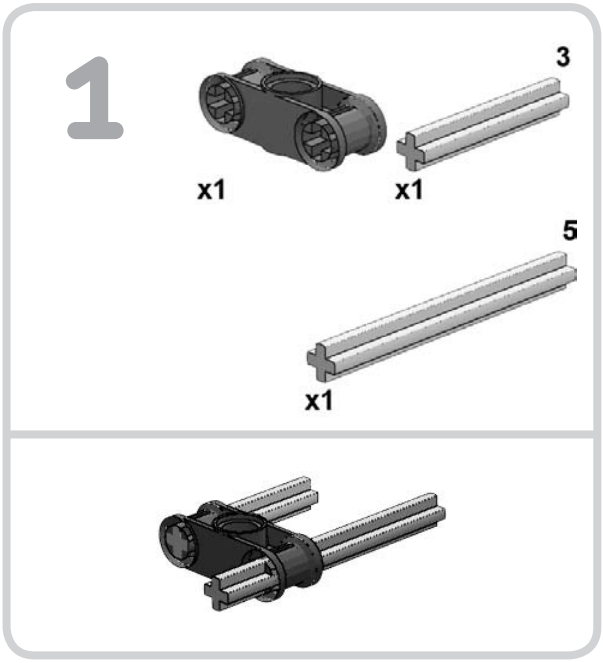


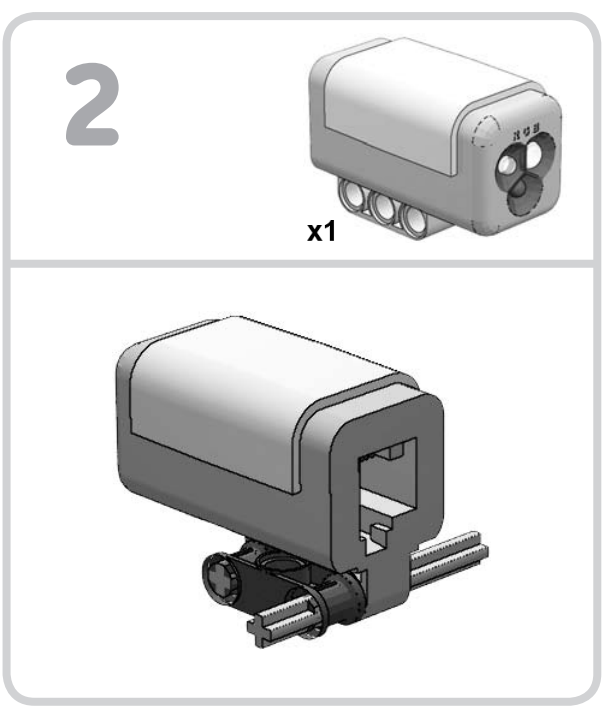
color sensor or light sensor

Follow the next steps to connect the Color Sensor or Light Sensor to the robot. The pictures show the Color Sensor from the NXT 2.0 retail kit; however, substitute the Light Sensor if you have another kit.

First create a mounting bracket for the sensor.



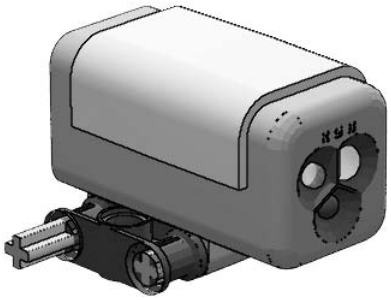




Turn the sensor around for the next step.

3

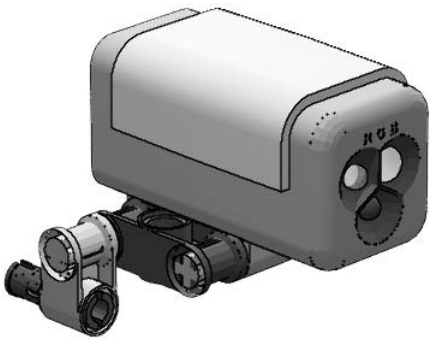
x1



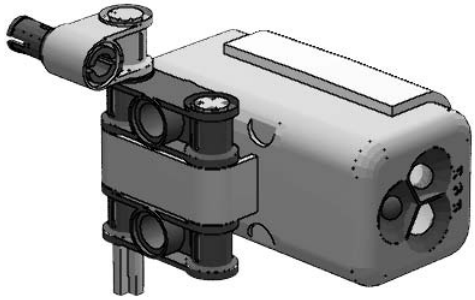
4

x1

x1



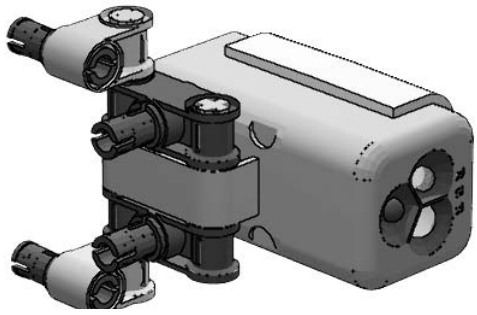
Turn the sensor on its side for the following steps.



5

x1

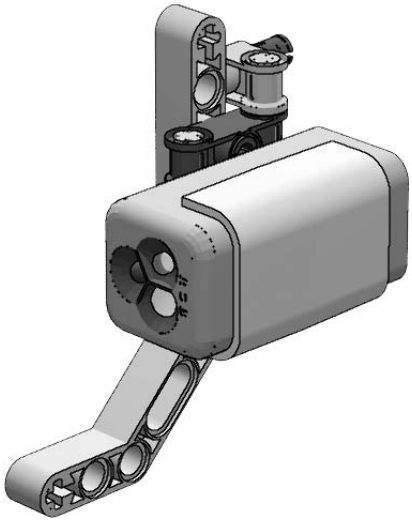
x3



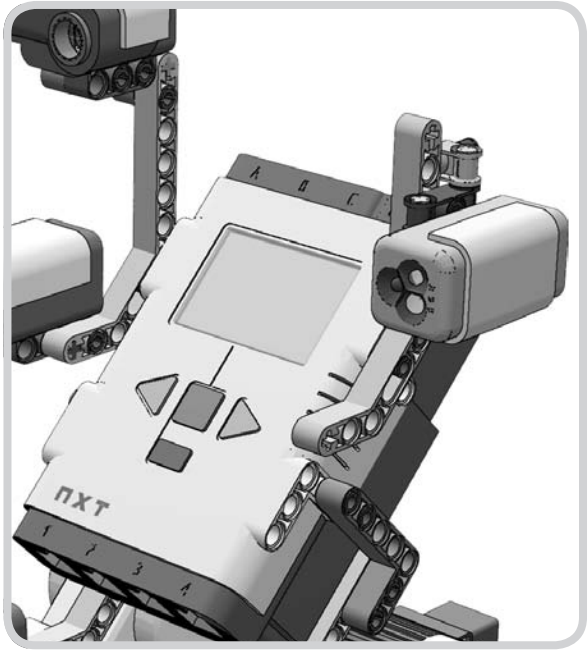
Next attach the sensor to a beam.

6

x1



Attach the beam to the side of the TriBot.



attach the wires

It's easier to add the wires now before you add the final beam. Table 3-1 shows how to connect the motors and sensors to the NXT.

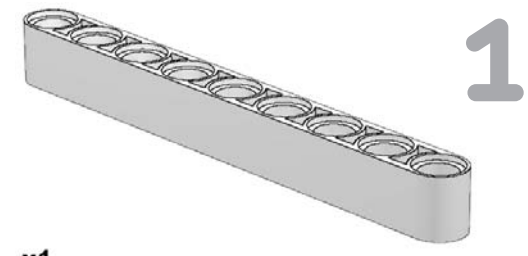
table 3-1: cable connections

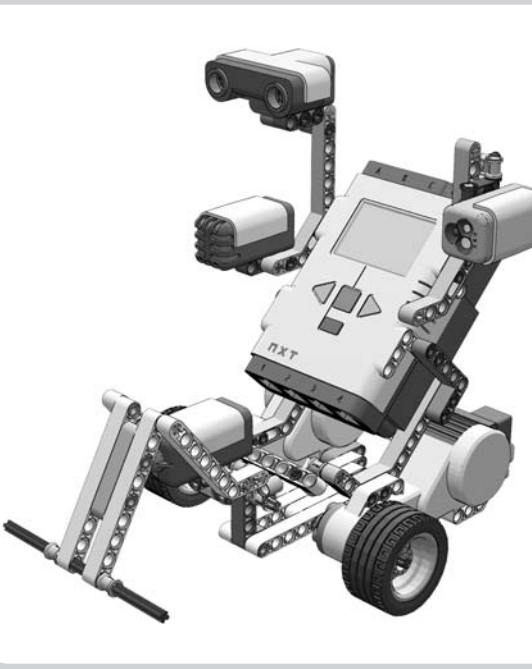
motor or sensor	port	cable length
Motor on the Ultrasonic Sensor side	B	14 inches (35 cm)
Motor on the Color/Light Sensor side	C	14 inches (35 cm)
Touch Sensor	1	8 inches (20 cm)
Sound Sensor	2	14 inches (35 cm)
Color or Light Sensor	3	14 inches (35 cm)
Ultrasonic Sensor	4	20 inches (50 cm)

The motors use ports B and C to match the default settings for the Move Block. Likewise, the NXT-G blocks that use sensors default to the port listed in the table. Any motor or sensor will work using any port (for example, the Touch Sensor works just as well using port 4 as it does using port 2), but using the default ports makes writing your programs a bit easier and less prone to error because you won't have to change the port setting. The programs in this book assume you have the motors and sensors connected according to Table 3-1.

the final beam

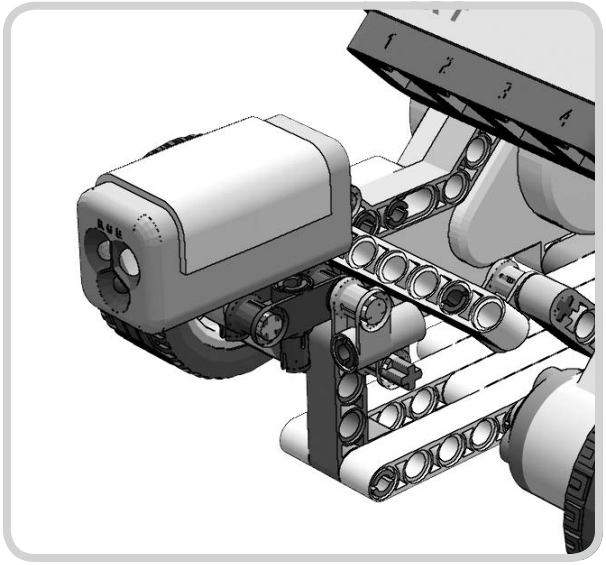
With the wires in place, it's time to add the last piece. Add this beam across the front of the robot, just behind the Touch Sensor.



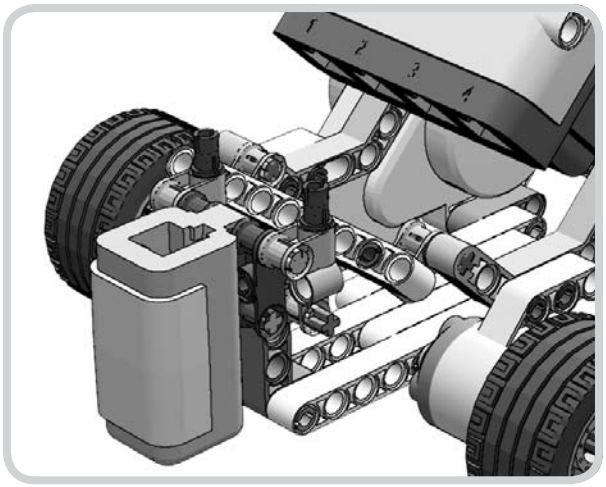


alternate placement for the color sensor

Some programs require the Color or Light Sensor to be placed at the front of the TriBot, replacing the Touch Sensor. The sensor can be mounted pointing forward as shown here:



You can also mount the sensor pointing down, as shown here:

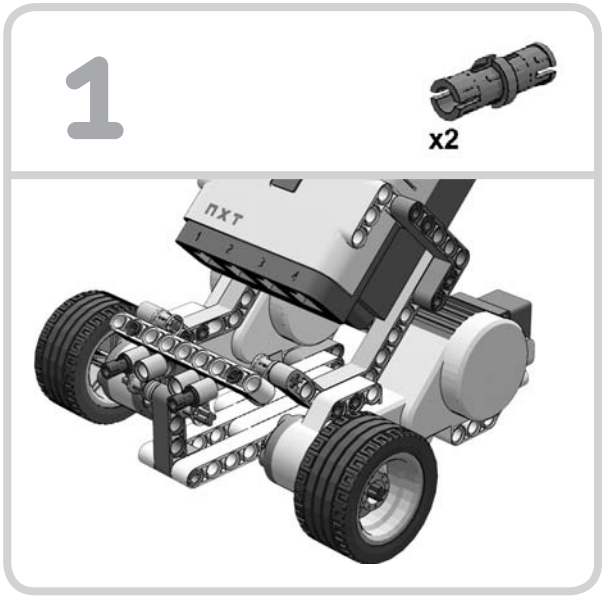


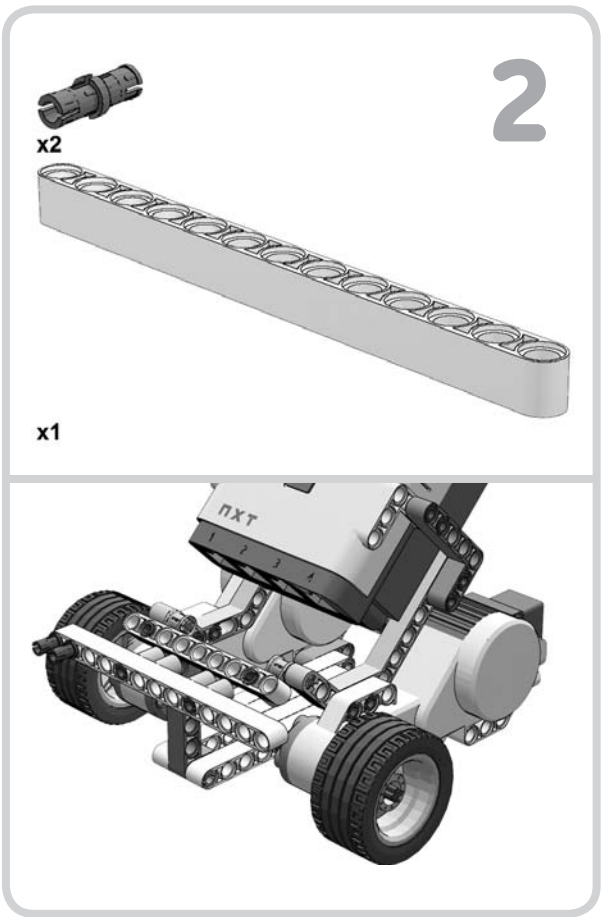
alternate placement for the ultrasonic sensor

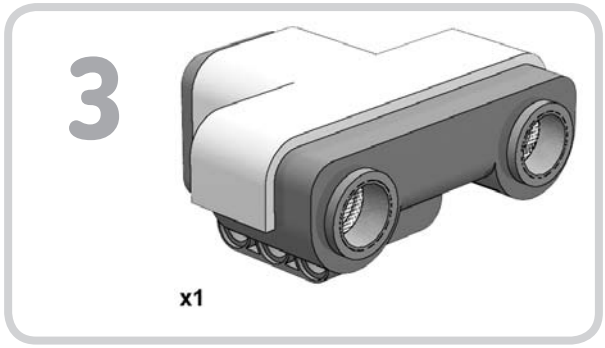
Some programs require placing the Ultrasonic Sensor so it points to the side of the TriBot. Follow these steps to make the necessary changes.

Start by removing the Ultrasonic Sensor and the Touch Sensor bumper.

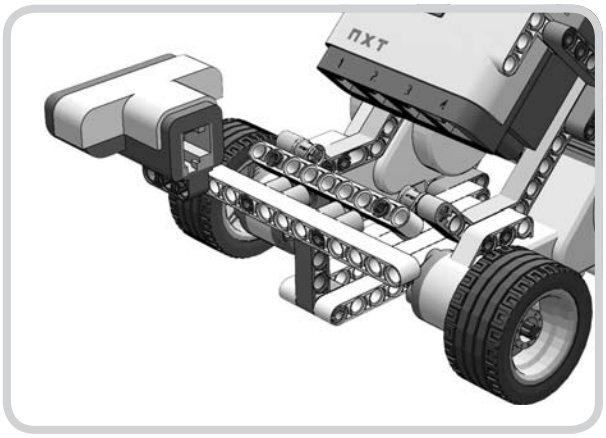
The next steps add a beam to the front of the TriBot and attach the Ultrasonic Sensor and the Touch Sensor bumper.



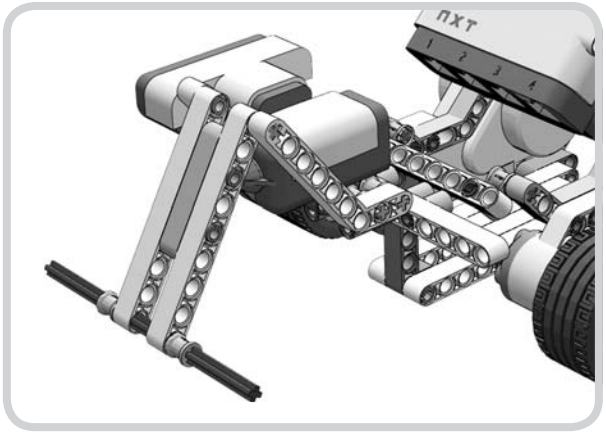




Add the Ultrasonic Sensor to the end of the beam.



Finally, put the Touch Sensor bumper back in its place.



conclusion

Now that you have built the TriBot, you can use it for the programs in the remainder of this book. The first programs will use the original configuration, with the Touch Sensor on the front. I'll let you know when to use one of the alternate sensor placements.

4

motion

The one thing above all else that excites people about working with robots is that they move. There is just something fascinating about watching your creation become mobile.

We can thank the NXT motors for making this movement possible. In this chapter, you'll learn about the NXT-G blocks used to control the motors, beginning with a very simple program and working up to some more complex examples.

the NXT motor

The NXT motor (shown in Figure 4-1) is designed to make it easy for you to create moving robots. The case has an unusual shape because it contains a set of gears in addition to the electric motor. The gears adjust the speed and power of the motor's rotation, making it possible to connect a wheel directly to the NXT motor without the need for additional gears.

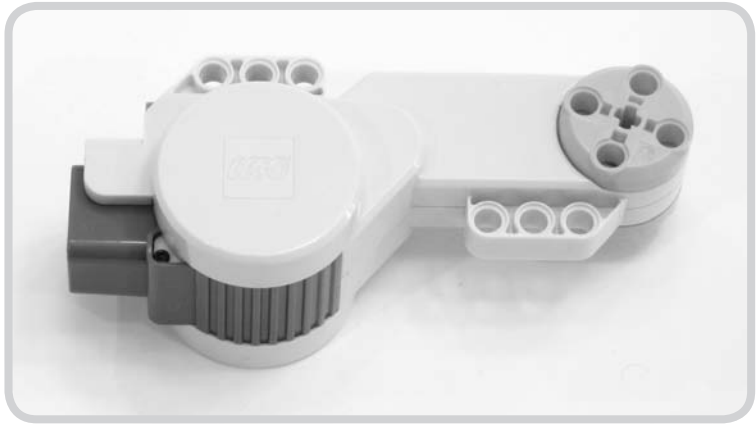


Figure 4-1: The NXT motor

Although the balance of speed and power provided by the NXT motor is great for a robot like the TriBot, it's not ideal for every situation. Your NXT kit comes with a collection of gears that you can use if your design requires more speed or power. *The Unofficial LEGO MINDSTORMS NXT Inventor's Guide* by David J. Perdue (No Starch Press, 2007) has an excellent section on using gears in NXT robots.

The NXT motor contains a built-in *Rotation Sensor* to measure how much the motor rotates. Because the sensor is part of the motor, you don't have to use one of the four sensor ports with a separate sensor. The NXT-G Move and Motor blocks (discussed in a moment) automatically use this sensor to make very precise moves. In addition, you can use this sensor in your programs to control the robot's movement. The next chapter discusses the Rotation Sensor along with the other sensors.

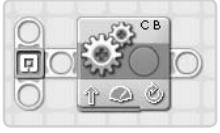
the move block

The Move block (shown in Figure 4-2) can control any of the three motors working alone, two of the motors working together, or even all three motors at the same time. This is the block you'll use most often to make your robot move. It has many options and is suited to a wide variety of tasks.

Figure 4-2: The Move block

For example, the following is a simple program to move the TriBot forward a short distance:

1. Create a new program named *SimpleMove*.
2. Drag a Move block onto the Sequence Beam from the top of the Common Palette. Leave all the settings with the default values. Your finished program should look like this:



3. Download and run the program. Your robot should move forward a few inches.

the move block's configuration panel

Although the Move block's default values work for the simple program you just created, you'll usually need to make some changes. The Configuration Panel (shown in Figure 4-3) contains all the options you need to customize the block's behavior. The settings you choose will depend on how you construct the robot and what you want it to do. The various sections of the Configuration Panel are explained here, followed by some example programs.

port

The Port setting (shown in Figure 4-4) is where you tell the NXT which motors you want to move. As you can see, you can select one, two, or all three ports.

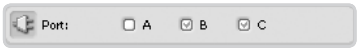


Figure 4-4: Selecting the port

It's common for a robot to use two motors to move. The TriBot works this way, and so do many other robot designs. When two ports are selected, the Move block automatically keeps the motors synchronized—in other words, it constantly adjusts how fast each motor moves so the two wheels work together. The Move block uses the Rotation Sensors in the motors to make the robot move in a straight line, turn a corner, or even spin in place, based on the setting of the Steering control (discussed later in this chapter). Having the Move block synchronize the motors immensely simplifies the task of controlling a robot's motion.

NOTE **Although it's possible to select all three ports, it's unlikely you will ever do so. Using all three motors together generally requires more coordination than a single block provides. If you do choose to select all three ports, only the B and C motors will be synchronized, and the Steering control becomes disabled.**

direction

For the Direction setting of the Configuration Panel, the choices are Forward, Backward, and Stop, as shown in Figure 4-5. The only thing you need to be careful of here is that Forward and Backward are relative to how your robot is constructed and oriented. Luckily, it's easy to tell when this setting is wrong (your robot will move backward when you want it to move forward), and the fix is just to select the other option. If only all bugs were this easy to detect and solve!

Figure 4-5: Setting the direction

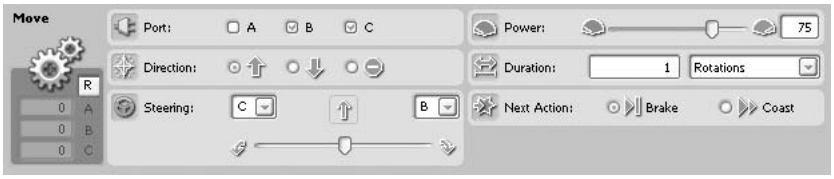


Figure 4-3: The Move block's Configuration Panel

The Stop setting is used to stop a motor that a previous Move block started.

power

The Power setting (shown in Figure 4-6) controls how fast the motors will move. Set the Power either by moving the slider or by typing a value from 0 to 100. A setting of 100 will make the motors move as fast as they can, and a motor won't move at all with the Power set at 0. It takes a minimum amount of power for a robot to being moving, depending on the robot's weight, the surface it's on, how the wheels are connected, and the strength of the batteries. The lowest setting I use to make the TriBot run across my desk is 7. Using a setting too close to the minimum can cause *stalling*—when the motor is unable to move either because it doesn't have enough power or because something is blocking it.

Figure 4-6: How fast the motor should move

When setting the Power item, you need to balance accuracy with speed. A slow-moving robot can move more accurately, but a faster-moving robot gets its job done more quickly, which is important in timed tasks. A fast-moving robot is also more fun to watch. Deciding on the best Power setting for your robot usually takes some trial and error.

duration

The Duration setting controls how long a move will last. There are four choices, as shown in Figure 4-7: Unlimited, Degrees, Rotations, and Seconds.

Figure 4-7: Setting the duration

The Degrees and Rotations settings let you set how much the motor will turn. The only difference between these two settings is the number that you enter—typing 360 degrees will turn the motor exactly one rotation. It's more convenient to measure a long move in rotations and a short move in degrees, because the numbers are easier to work with, just like choosing between using feet and inches (or meters and centimeters).

When entering numbers in degrees, you can use whole numbers only, but when using rotations, you can use up to three decimal places. However, the Move block is only accurate to within one degree, so using rotations won't result in better accuracy even though you can enter more precise numbers.

Here are a few details you should know about when using the Degrees and Rotations settings:

- Using negative numbers will not move your robot backward. You need to set the Direction item to Backward to do this.
- A Power setting that is too low to move the motor can cause your program to stop. The program will not start the next block until the motor moves the complete distance. This means that if the motor stalls, the block will never finish. For example, if you tell the motor to move 300 degrees with the Power set to 2, then because the robot doesn't move, it will never reach 300 degrees and so the program never moves to the next block (actually, you can force the block to finish, but it requires using another Move block running on a second Sequence Beam; Chapter 17 discusses using multiple Sequence Beams in detail).
- When you switch between Degrees and Rotations (or the other way around), the number you enter will change to match the new setting, keeping the distance the same. For example, if you set the Duration to 2 rotations and then select Degrees, the number will change to 720. If you want to change both the number and the unit, make sure to change the unit first; otherwise, you'll end up with the wrong distance. To go from 2 rotations to 500 degrees, select Degrees first and then enter 500. If you enter the 500 first and then select Degrees, the 500 will change to 180000 degrees (which is 500 rotations).

Set the Duration item to Seconds to run the motors for the length of time you specify. You can use up to three decimal places when setting the value, just like the Wait Time block.

An Unlimited move will keep the motors going until you run another Move block or the program ends. In most cases, a block will finish what it's doing before the next block on the Sequence Beam runs. A Move block with the Duration set to Unlimited is one of the exceptions (because the move will never complete by itself). With any of the other Duration choices, your program will wait until the move is complete before starting the next block.

steering

Use the Steering slider (see Figure 4-8) to control how quickly your robot turns or to make it move in a straight line. You can set the Steering option only if you select two ports—this control is disabled if you select one or three ports.

Figure 4-8: Steering control

Setting the slider in the middle makes the robot move straight—at least, the NXT will *try* to make it go straight by constantly making small adjustments to the motors to keep them moving at the same speed. Many things can affect how a robot moves, and your program can only control how fast it moves each motor. If your robot is *unbalanced*, meaning that there is more weight on one side than the other, it will tend to drift to one side. The type of floor on which your robot is moving will affect its motion, as well as the wheels and the type of caster (third wheel) you use. It's almost impossible to make your robot move perfectly straight, but you can get close enough for most situations.

Setting the slider all the way to one side or the other makes a robot spin around, because the two motors will move at the same speed but in opposite directions. The distance between the two wheels determines the duration you need to set to make your robot spin in a full circle.

Setting the Steering slider somewhere between the middle and one end causes a robot to make a gentle turn. The closer you are to the end of the slider, the tighter the turn will be. Turning happens when one motor moves faster then the other; on the inside of the curve, the slower motor will move a shorter distance. If you select Degrees or Rotations for the Duration, then this setting will apply to the faster-moving motor, which is the one on the outside of the curve.

next action

The Next Action item tells the block how to end the move and what to do with the motor once the move is complete. The choices are Brake and Coast (see Figure 4-9). The Brake option quickly stops the motor and locks it in place. The Coast option lets the motor come to a stop on its own and then lets the motor spin freely.

Figure 4-9: How to end the move

The following are some things you should know about the Next Action item:

* You can only specify a Next Action if the Duration is set to Seconds, Degrees, or Rotations. If you set the Duration to Unlimited, this setting will be disabled because the block will finish while the motor is still moving.
* If you want your robot to make an accurate stop, use the Brake setting when the Duration is in Degrees or Rotations. The motor will slow down and stop very close to the Duration you set. With the Coast option, the motor will start to slow down after reaching the Duration, so it will move a little past the Duration you set.
* Use Brake to keep the motor from moving after the block finishes; for example, use Brake to hold the motor still after grabbing an object to keep it from falling out of a gripper.
* Holding a motor in place uses a small amount of battery power, so it's a good idea to set a motor to Coast unless you need to hold it in place.

the feedback boxes

Deciding on the value for the Duration can be a time-consuming task. The *Feedback Boxes*, located on the right side of the Configuration Panel and shown in Figure 4-10, can be a big help. They show how far each motor has moved. The value is always displayed in degrees (even if you set the Duration item to one of the other choices). For the Feedback Boxes to work, you need to have the MINDSTORMS environment connected to the NXT using either a USB cable or a Bluetooth connection.

Figure 4-10: The Feedback Boxes

The value will be red if the motor moved backward. Only the total distance is displayed, so if you move the motor forward 360 degrees and then backward 360 degrees, the display will show 0. The values change only for the motors selected in the Ports area of the Configuration Panel. Click the *R* button to reset the values to 0.

When the NXT is not running a program, you can move a motor manually and see how many degrees it moves. Then use the value displayed in the Feedback Boxes to set the Duration for your Move block. This is particularly useful when you need to determine how far to move an arm or gripper.

When you run a program, the values will reset to 0 at the start, will update while the program is running, and will reset to 0 again when the program completes. You can add a Wait Time block at the end of the program to make it pause if you want to read the values before they reset.

the NXT intelligent brick view menu

The Feedback Boxes are very convenient, but there are two drawbacks to using them: You need to keep the NXT connected to the computer, and you need to be sitting in front of your computer to see them. As an alternative, you can measure how far a motor moves just using the NXT.

Using the menu on the NXT, select View, Motor rotations, or Motor degrees, and then set the Port item (to either A, B, or C). When you move the motor attached to the selected port, the NXT's display will show how far it moves.

there and back

The ThereAndBack program will make the TriBot move forward 3 feet, turn around, and then return to where it started. Some measuring is required to solve this problem, so if your rulers are metric, feel free to change the problem to use 1 meter instead of 3 feet.

This program uses three Move blocks: one to move forward, one to spin the robot around, and one to move the robot back to where it started.

moving forward

The first block needs to move the robot forward 3 feet. "Feet" is not one of the duration options, so you need to figure out how far 3 feet is in either degrees or rotations—it doesn't matter which you choose. I'll use rotations for this example.

How do you figure out how many rotations you need to turn the motor to travel 3 feet? One way is to write a program that moves the robot a long distance, say 10 rotations. Before running the program, mark your robot's starting position. Run the program, and then measure how far the robot moved in inches. Divide the distance by the number of rotations to determine how far the robot moves in one rotation. My robot moves 52 inches in 10 rotations, or 5.2 inches in a single rotation. The robot needs to go 36 inches, so I need to divide 36 by 5.2 to find the proper Duration setting. I'll start with 6.9 rotations and adjust the value after some testing.

Be aware that the number you need will be slightly different if you change the type of floor you run on, the wheels you use, or the Power setting for the Move block. After some testing, I found that a Duration of 7.1 rotations and a Power setting of 75 works well for me.

NOTE **The balloon tires from the education set and the original NXT retail kit are larger and therefore go farther in one rotation. A duration of 5.25 works when using these tires.**

Once you know the duration, you can start writing the program. The following is the first set of steps:

1. Create a new program called *ThereAndBack*.
2. Drop a Move block onto the Sequence Beam, and set the Duration to the number you decided on. For now, keep the Power at the default value of 75.

Figure 4-11 and Figure 4-12 show the program and the Configuration Panel for the Move block.

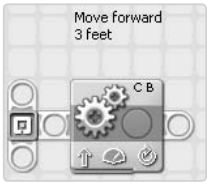


Figure 4-11: Step 1 of ThereAndBack

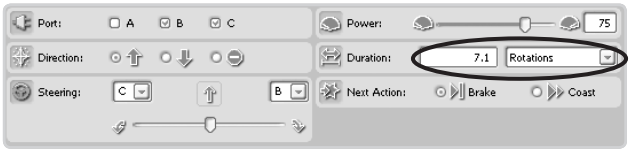


Figure 4-12: Setting the Duration item to 3 feet

Test these settings by running the program several times across a measured distance. Your robot should stop very close to the same spot each time. If it doesn't, you should try lowering the Power setting. Adjust the Duration if the robot stops at the same spot but doesn't travel the correct distance.

turning around

A second Move block will turn the robot around for the return trip. Moving the Steering slider all the way to one side makes the TriBot spin in place. It doesn't matter which side you pick; the direction the robot spins isn't important.

The only real challenge configuring this block is that you once again need to figure out what to use for the Duration item. After some testing I found that a Duration setting of 475 degrees works well for me (400 degrees using the balloon tires). I turned the Power down to 50 to get a good, consistent turn, because with the Power setting at 75, it turned a little too far about half the time. This is a typical trade-off between speed and accuracy.

The following are the next steps in building this part of the program:

3. Drag a Move block onto the Sequence Beam to the right of the existing block.
4. Drag the Steering slider all the way to one side.
5. Set the Duration to **475 degrees** and the Power to **50**. Use these for initial values; you can adjust them as needed during testing.

Figure 4-13 and Figure 4-14 show the program and the configuration of the second Move block.

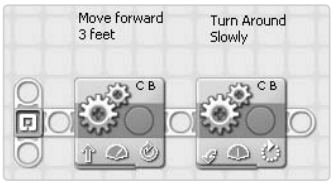


Figure 4-13: Moving forward and turning around

Figure 4-14: Configuration Panel to turn around

testing a single block

To fine-tune the Duration and Power settings, you can test this block on its own. It's easier to test the block over and over by itself instead of waiting for the robot to travel 3 feet and then seeing whether it turns around correctly. Test a single block using the Download and Run Selected button on the Controller (shown in Figure 4-15). If you select the second Move block and then click the Download and Run Selected button, your robot will only turn around (instead of moving the first 3 feet). Adjust the Duration and Power settings as needed until your robot consistently turns all the way around.

Figure 4-15: Download and Run Selected button

moving back to the start

To move the TriBot back to where it started, add a third Move block that travels the same distance as the first Move block. The final steps in the program are as follows:

6. Drag a Move block onto the end of the Sequence Beam.
7. Set the Duration to the same value you used for the first Move block.

Figure 4-16 shows the final program.

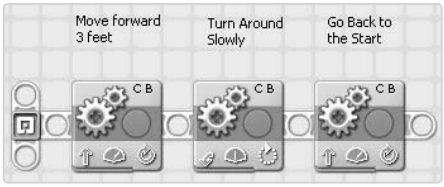


Figure 4-16: Final ThereAndBack program

Test your program with all three blocks. A slight error in the Duration when turning around may show up after traveling back the 3 feet, so you may need to make an adjustment to the Power or Duration setting of the second Move block.

around the block

The next program will make the TriBot travel around a square and stop where it started. For this example, I'll use a square that is three rotations long on each side. At the corner, the robot will move in a gentle curve rather than just spinning in place. The robot doesn't need to hug the edges of the square, just travel around it, and you can get a smoother motion by using a curve instead of spinning.

To travel around a square, the robot needs to move along a side and go around the corner, move along the next side and go around that corner, continuing this for all four sides.

the first side and corner

The first part of the program uses two Move blocks to move the TriBot along the side and make the first turn. Moving along the edge is easy; just set the Duration to three rotations. Moving around the corner requires setting both the Steering and Duration items. Set the Steering item to about three quarters of the way between the middle and end of the slider. The next step is to find the Duration setting that gives you an accurate turn around the corner. After some experimentation, I found that 5.3 rotations works well for me (4.1 using the balloon tires). Once again, your setting may be a little different because it depends on many factors, including the exact Steering setting and the surface you use. The following are the steps to build this part of the program:

1. Create a new program called *AroundTheBlock*.
2. Drag a Move block onto the Sequence Beam.
3. Set the Duration item to **3 rotations**.
4. Add a second Move block to the program.
5. Set the Duration item to **5.3 rotations**.
6. Drag the Steering slider toward the end of the slider.

Figure 4-17, Figure 4-18, and Figure 4-19 show the program and the Configuration Panels for the two Move blocks.

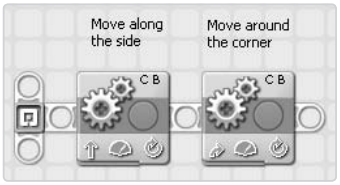


Figure 4-17: Moving along the side and around the corner

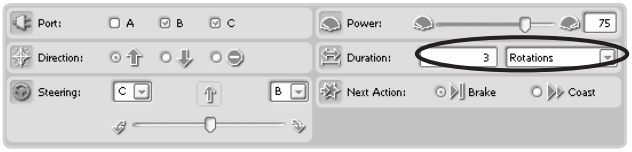


Figure 4-18: Moving along the side

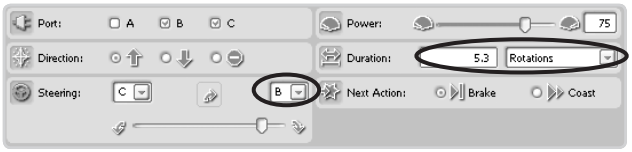


Figure 4-19: Moving around the corner

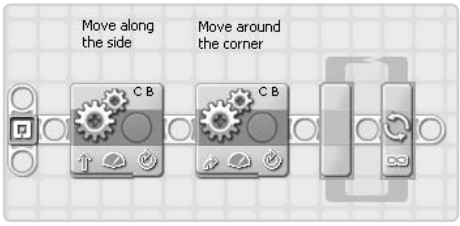
the other three sides and corners

The next step is to extend the program to go all the way around the square. You could add six more Move blocks and use the same settings for the other three edges and corners. That would be a little tedious, and imagine if you wanted to go around the square 10 times—you would need to add 78 more blocks! Of course, there is an easier way, and it involves the Loop block.

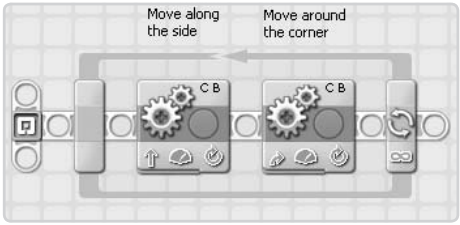
The Loop block (shown in Figure 4-20) lets you run a group of blocks more than once. You can run the two Move blocks four times (once for each side of the square) by placing them on the Sequence Beam inside a Loop block. There are several ways to control the Loop block, and you'll use it often. For now, I'll only cover the settings you need to use for this program and go into more detail in Chapter 6.

Figure 4-20: The Loop block

7. Drag a Loop block onto the end of the Sequence Beam from the Common Palette. Your program should look like this:



8. Drag the two Move blocks to the middle of the Loop block. The Loop block will expand as you drop in the Move blocks. Now the program should look like this:



NOTE **If you added comments above the two Move blocks as I did, then you should also move the comments so they stay above the blocks they describe. Comments don't automatically move with the blocks, so you may want to add them after you finish editing the program.**

The blocks in the Loop need to run four times, once for each side. Figure 4-21 shows the Loop block's Configuration Panel with the changes you need. The following are the steps to make the loop execute four times:

9. Select **Count** from the list of options for the Control item.
10. Enter **4** for the Count item.

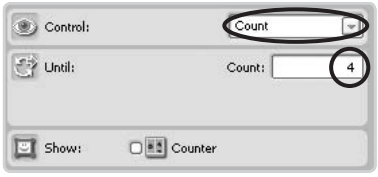


Figure 4-21: Going around the loop four times

Figure 4-22 shows the complete program. Notice that the small infinity sign that was at the bottom of the right side of the Loop block changed to a tiny abacus (at least I think it's an abacus—it looks like one, and an abacus would make sense with the Control item set to Count). The icon lets you know how the Loop block is configured without looking at its Configuration Panel.

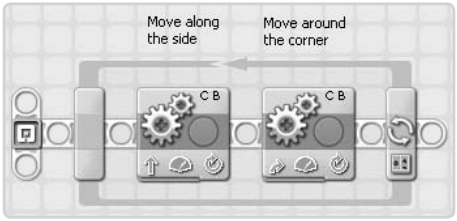


Figure 4-22: The complete AroundTheBlock program

testing the program

When you run the program, your TriBot should move all the way around the square. If your robot doesn't end up where it started, adjust the Duration for the second Move block. The "turning" block runs four times, and any error will accumulate as the robot moves around the square. There will always be some error; no mechanical system is exact. The goal is to get the error to be small enough not to impact your program too much, which means getting reasonably close to the starting position.

the motor block

One nice thing about the Move block is that it handles all the details involved in controlling a motor. You can set the duration to six rotations, and the block will take care of bringing the motor up to speed, moving at the power level you selected, and then slowing down and stopping at just the right distance. This works great almost all of the time, but in some situations, you may want more control over a motor. For example, the Move block may speed up too quickly, or the synchronization between two wheels may not be good enough for certain surfaces or robot designs.

The Motor block controls a single motor and lets you choose how quickly the motor *accelerates* (speeds up) at the beginning of a move and how quickly it *decelerates* (slows down) at the end. The Motor block does not appear on the Common Palette; you have to select it from the Complete Palette. As shown in Figure 4-23, it's in the Action group, which is the second group from the top of the palette. When you add a Motor block to your program, it is displayed in the Work Area, as shown in Figure 4-24.

Figure 4-23: The Motor block on the Complete Palette

Figure 4-24: The Motor block

When you move a motor, the motion consists of the following three parts or phases:

1. During the *ramping up* phase, the motor accelerates to the Power level you set.
2. During the *constant* phase, the motor keeps moving at the same speed.
3. During the *ramping down* phase, the motor decelerates to a stop (or a lower Power level).

You can configure a single Motor block to perform any one of the three phases. By using three Motor blocks (one for each phase), you can get very fine control over a motor.

The Motor block's Configuration Panel (shown in Figure 4-25) is very similar to the Move block's, and the Direction, Power, Duration, and Next Action items behave the same for both blocks.

The real power of the Motor block is in the Action item. There are three choices: Ramp Up, Constant, and Ramp Down, corresponding to the three phases mentioned earlier. The Ramp Up choice lets you control how quickly the motor accelerates to the Power setting you set. The Duration setting is used to control how long to take to bring the motor up to the selected Power level and can be set in seconds, degrees, or rotations.

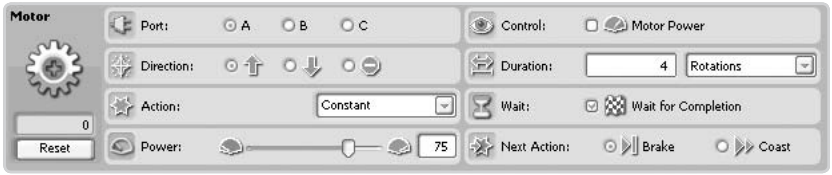


Figure 4-25: The Motor block's Configuration Panel

Similarly, the Ramp Down option lets you control how quickly the motor decelerates. Set the Power item to the setting you want at the end of the move. So to stop the motor, set Power to 0. The Duration setting determines how long it will take to decrease the speed of the motor from its current level to the target Power setting.

Figure 4-26: The CoastTest program

The Constant option keeps the motor moving at the levels you set for the Power level and Duration. If the motor is not at the specified Power level when the block starts, it will very quickly change the motor's speed, with very little ramping.

To have complete control over the motion, use three Motor blocks together, one for each of the Action choices. Set the Next Action item to Coast for the first two blocks; otherwise, the motor will come to an abrupt stop at the end of each block instead of smoothly transitioning to the next block.

The following are a few other differences between this block and the Move block:

- You can select only one port. Unlike the Move block, the Motor block can control only one motor, which is why there is no Steering control and only one Feedback Box.
- The Motor Power option will try to keep the motor spinning at the same rate by increasing the power level if the motor starts to encounter some resistance.
- The Wait for Completion option lets you run the next block in the program immediately or wait until the move is complete. The Move block doesn't have this option; it always waits until the move is complete before continuing except when the Duration item is set to Unlimited.

Some programmers prefer to use the Motor block rather than the Move block when controlling a single motor, while others use it only when the Move block doesn't work well enough for a particular situation.

Figure 4-27: Configuration Panel for the first two Move blocks

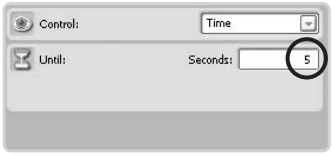


Figure 4-28: Configuration Panel for the Wait Time blocks

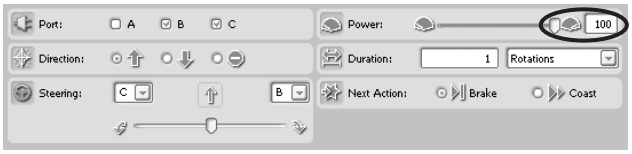


Figure 4-29: Configuration Panel for the final Move block

brake, coast, and the reset motor block

The Move and Motor blocks both have a Next Action item, allowing you to specify what to do at the end of the move. As you saw earlier, the choices are Brake and Coast. When the Coast setting is combined with a Duration of either Rotations or Degrees, these blocks behave in a way that many users find unexpected: After coasting to a stop, the next move seems to be a little short or a little long.

When you run a Move or Motor block using these settings, the firmware will keep track of how far the motor actually moved (the firmware is the program that runs on the NXT and executes the program you write). Because the motor coasts to a stop, it will move a little more than the Duration setting. The next Move or Motor block to run will adjust its Duration to account for the extra distance.

the CoastTest program

The CoastTest program, shown in Figures 4-26 through 4-29, demonstrates this behavior. This program has three Move blocks with the Duration set to 1 rotation, for a total of three rotations. The Next Action item is set to Coast for the

first two blocks and Brake for the final one. The Power is set to 100 to maximize the effects of coasting.

It's a little difficult to see the effect of the Coast setting by just running this program. Putting a small sticker on one of the wheels makes it clear how far the wheel has moved with each block. Table 4-1 shows the motor position, and Figures 4-30 through Figure 4-33 show the position of the wheel at the start of the program and after each block.

table 4-1: motor positions for the CoastTest program

program point	motor position in degrees
Starting position	0
After first Move block	405
After second Move block	758
Final position	1080

Figure 4-30: Starting position

Figure 4-31: After the first Move block

Figure 4-32: After the second Move block

Figure 4-33: The final position

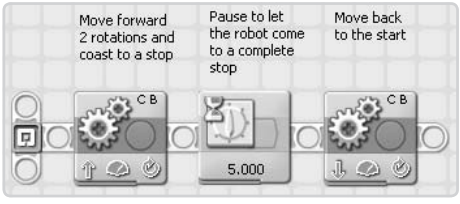


Figure 4-34: The CoastBack program

Notice that the first Move block went a full rotation and then an extra 45 degrees, for a total of 405 degrees. The second move went only 353 degrees, a little less than a full rotation. Two full rotations is 720 degrees, so after the second move, the robot is still 38 degrees beyond the combined Duration setting for the two Move blocks. The final move stops at the correct position, which is at three rotations, or 1,080 degrees.

The important thing to notice is that the total duration of the three Move blocks is three rotations, which is how far the robot moved. The firmware compensated for the extra distance that the motors coasted so that the total distance traveled by all three block would be accurate.

a problem with coasting

Having the firmware account for the distance the robot coasts is not always the desired behavior. The CoastBack program, shown in Figures 4-34 through 4-37, is a typical example of the problem. The two Move blocks have the Duration item set to 2 rotations and the Next Action item set to Coast; the only difference between the two blocks is the Direction setting. The program moves the robot forward two rotations and waits for the robot to coast to a stop. Then it moves the robot backward to the starting point, coasting to a stop again. The expectation is that the TriBot will finish in the same place it started because the two Move blocks have identical settings (except for the Direction setting).

Figure 4-35: The first Move block

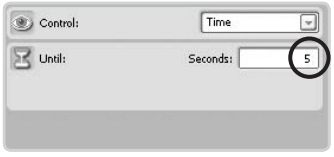


Figure 4-36: The Wait Time block

Figure 4-37: The second Move block

observing the problem

When this program is run, the TriBot won't stop in the same place it started. It moves past the starting point and stops a small distance away. If you watch closely, you may notice that the robot starts slowing down just as it gets to the starting point, instead of slowing down earlier and coasting to the starting point. Figure 4-38 and Figure 4-39 show the starting and ending positions of a test run.

Figure 4-38: The starting position

Figure 4-39: The final position

The robot didn't stop at the same place it started because the firmware adjusted the duration of the second Move block to account for the extra distance the first Move block coasted. To make the robot stop the motors after exactly two rotations and coast back to the starting point, tell the firmware that you don't want it to make the adjustment.

the reset motor block

To be technically correct, NXT-G doesn't have a way to prevent the firmware from making the adjustment. However, you can reset the adjustment value to zero using the Reset Motor block (shown in Figure 4-40), which has the same effect. The Configuration Panel (shown in Figure 4-41) is very simple; you just select the motors you want reset.

Figure 4-40: The Reset Motor block

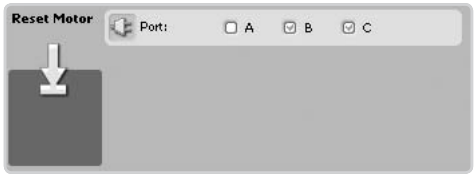


Figure 4-41: Configuration Panel for the Reset Motor block

Adding a Reset Motor block before the second Move block fixes the CoastBack program (see Figure 4-42). The default value for the Port setting is B and C, so you don't need to make any changes to the block's configuration. Now when the program is run, the TriBot will move forward two rotations and coast to a stop and then move backward two rotations and coast to a stop, which will put it very close to the starting point.

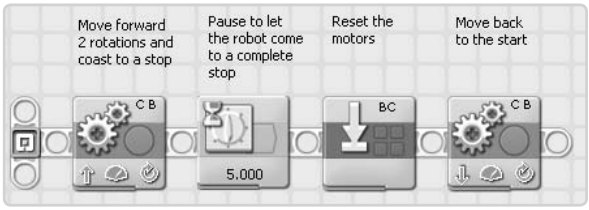


Figure 4-42: Final CoastBack program

the record/play block

The Record/Play block (shown in Figure 4-43) lets you record the movements of one or more motors and later play them back, reproducing the movements you recorded.

Figure 4-43: The Record/Play block

Using this block is a two-step process. First, you write a program with the block's Action item set to Record. While running the program, manually move the robot through the actions you want it to perform. The block will save a file on the NXT containing the information needed to replay the movements.

The second step is to use a block with the Action item set to Play in another program to replay the motions you recorded.

configuration panel

The Configuration Panel contains different items depending on whether the block is used to record or play. Figure 4-44 shows the Configuration Panel with Record Action selected. The value entered in the Name section is used to create the file on the NXT that will hold the movement information. The other items allow you to select which motors you are interested in and the number of seconds to record.

Figure 4-44: Configuration Panel for recording a motion

When you select Play for the Action item, the Configuration Panel will appear as shown in Figure 4-45. To play back a recorded motion, give the name that was used when the motion was recorded. The Files area lists any previously recorded motions that are on the NXT.

Figure 4-45: Configuration Panel for playing a motion

The Record/Play block is useful when you have a complex series of motions that you need to repeat, for example, a complicated path that a wheeled robot needs to follow or the movements of a robotic arm using multiple motors. It can be a little difficult to record the motion the way you want because moving the motors by hand is generally not as smooth as using a program, and it may take a few tries before you are satisfied with the result. Another thing to keep in mind is that the motion file stored on the NXT takes up some of the limited memory you have available. Managing the NXT's memory is covered in Chapter 12.

the remote control tool

The LEGO MINDSTORMS NXT 2.0 retail kit includes a remote control tool that lets you directly drive your robot from the MINDSTORMS software. Selecting the Tools ▸ Remote Control menu item opens the Remote Control window (shown in Figure 4-46). The four arrow buttons let you move your robot forward or backward and turn to the left or right, and the Action button controls the third motor (if your robot uses three motors). See the help file for a complete description of this handy little tool.

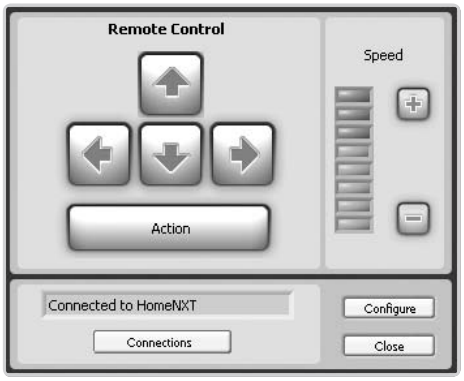


Figure 4-46: The Remote Control tool

conclusion

The NXT kit comes with three motors specifically designed to make it easy to build a wide variety of robots. The NXT-G language provides three blocks for controlling the motors, giving you lots of flexibility when deciding how your robot should move. The Move block is the most common choice because it's simple to use and its ability to synchronize two motors makes it easy to program a two-wheeled robot. The Motor block gives you more control when using a single motor, and the Record/Play block lets you record and then play back your robot's movements.

The example programs show a few different ways to use the Move block, including one that moves the TriBot around a square. At this point, you should be able to use several Move blocks together to program your robot to follow any course you design. Many of the example programs in the following chapters involve moving the TriBot, so you'll get plenty of practice using the motors.

5

sensors

In this chapter, you'll learn how to use the NXT sensors to get your robot to react to what is happening around it. In the example programs in the previous chapter, the robot didn't react to anything around it; it simply followed the path you programmed. By using the NXT sensors in your program, you can make your robot avoid obstacles, follow a line on the floor, react to light or sound, and identify objects based on their color.

We learn about the world around us using our five senses (touch, sight, sound, smell, and taste). A robot uses sensors in the same way to gather information about its environment. The LEGO MINDSTORMS NXT kits come with several sensors, each of which is useful for solving certain problems, depending on the kind of information it collects about the robot's environment.

In most cases, your program will use the data from the sensors to make decisions about what to do next, but it can also use sensors to collect data, usually as part of an experiment. This chapter covers the basic operation of each sensor and how to use the sensors to make decisions. Chapter 16 covers data collection.

using the sensors

Three NXT-G blocks have built-in support for sensors: the Wait, Loop, and Switch blocks. With these three blocks, you can make the program wait until something happens, run a group of blocks over and over until something happens, or choose which blocks to run based on the data from a sensor.

You've already seen how to use the Wait and Loop blocks to make the program pause or repeat a group of blocks. You can use the Switch block to choose between two or more groups of blocks, meaning your program can make decisions about which actions to perform. You'll use all three blocks for the programs in this chapter, and I'll cover the Loop and Switch blocks in detail in Chapter 6.

Figure 5-1 shows the Configuration Panel for the Wait block with the Control option selected. In Chapter 4, you set the Control item to Time to make the block wait for several seconds. The Wait, Loop, and Switch blocks all have a Control item that lets you choose what the block reacts to.

Figure 5-1: The Wait block Configuration Panel

To work with a sensor, set the Control item to Sensor and then select the sensor to use. The Configuration Panel should then change to include the settings for the selected sensor. The first program in this chapter uses the Wait block with the Touch Sensor so you can see how this works.

In addition to the three blocks mentioned previously, you'll see a separate block for each sensor. For example, you can use the Touch Sensor block to get data from the Touch Sensor. The sensor blocks don't take any action based on the data from the sensors; they just make the data available to other blocks by using data wires. Because data wires are the subject of Chapter 8, I'll put off the discussion of these blocks until then.

the touch sensor

The Touch Sensor (shown in Figure 5-2) has a small button on the front. NXT-G blocks can use input from this sensor to tell whether the button is pressed, released, or bumped (*bumped* means the button was pressed and then quickly released).

Programmers often use the Touch Sensor to detect when the robot has run into something, but you can also use it to control your program. For example, you can have the robot wait until you press the Touch Sensor before it starts moving.

Figure 5-2: The Touch Sensor

configuration panel

You can configure the Touch Sensor as part of the Wait, Loop, or Switch block. The part of the Configuration Panel that works with the Touch Sensor is identical for all three blocks. Figure 5-3 shows the Configuration Panel for a Wait block using the Touch Sensor.

The Configuration Panel has three sections. On the left is a single Feedback Box, similar to the one for the Move block. In the center are the options for the Wait block. The section on the right will change depending on the Control and Sensor choices you select. In this example, you can see the two options for the Touch Sensor.

The only two options to set for the Touch Sensor are the port the sensor is plugged into and the action you want to wait for. The four sensor ports on the bottom of the NXT are 1, 2, 3, and 4. Be sure to plug the sensors into one of these ports and not the motor ports at the top of the NXT (labeled A, B, and C). By default, port 1 is used for the Touch Sensor.

The Action setting defines the *trigger*—what you want to wait for—for the Wait block. You can trigger the block when the button on the Touch Sensor is pressed, released, or bumped.

feedback box

All the blocks that work with sensors have a Feedback Box so you can see the sensor's value. This is extremely helpful when deciding on a trigger value to use. The meaning of the value displayed depends on the sensor and how it's configured.

NOTE **Remember that the NXT must be turned on and connected to the MINDSTORMS IDE using the USB cable or via a Bluetooth connection for the Feedback Box to work.**

As far as the Touch Sensor is concerned, the meaning of the value displayed in the Feedback Box depends on the Action setting as follows:

- **Pressed.** The Feedback Box will show a *1* when the button is pressed and a *0* when the button is not pressed. (*1* and *0* are often used in computer programming to represent yes/no or true/false values.)
- **Released.** The values displayed are the opposite of those used for the Pressed option. A *1* is displayed when the button is not pressed, and a *0* is displayed when the button is pressed.
- **Bumped.** The Feedback Box will show a count of how many times the button has been bumped. You can reset the count to 0 by selecting one of the other actions and then reselecting Bumped.

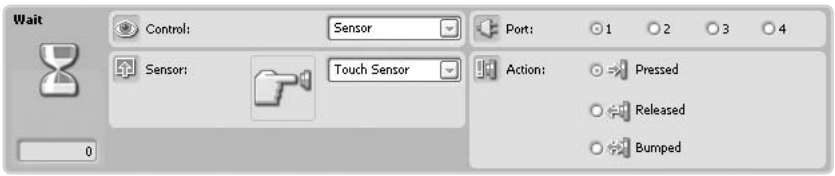


Figure 5-3: Waiting for the Touch Sensor to be pressed

the NXT's view menu

When working with the motors in Chapter 4, you saw that you can use the View menu on the NXT as an alternative to the Feedback Boxes. You can also use the View menu to display the value from a sensor; the menu contains options for each sensor. The meaning of the value displayed depends on the sensor you select.

For the Touch Sensor, the NXT will display a *1* when the button is pressed and a *0* when the button is released. This is a little simpler than the way the Feedback Box works, where the value displayed depends on the Action setting of the block. The Feedback Box can also show you if the sensor has been bumped, whereas the View menu can show only that it's pressed or released.

the BumperBot program

Now that you know how the Touch Sensor works, let's put it to use. In this section, you'll build the BumperBot program, which uses the bumper on the front of the TriBot to help the robot wander around a room. When the robot runs into something, the Touch Sensor will be pressed. The program will react by making the TriBot back up, turn around, and then start again. The robot keeps going until you stop it by pressing the Exit button on the NXT.

When the BumperBot program runs, the robot should keep moving in a straight line until it runs into something. A Move block with the Duration item set to Unlimited will move the robot forward until its movement is stopped by another Move block. A Wait block using the Touch Sensor tells the robot when it has run into something.

Once the Touch Sensor is pressed, the program should stop the motors and then make the TriBot back up a bit and turn in a different direction. Once the robot has turned away from whatever it bumped, it should start moving again until it runs into another obstacle. You'll place the whole program in a Loop block so the TriBot will keep going until you stop it.

You'll build this program in three sections; follow these steps to complete the first part of the program:

1. Create a new program named *BumperBot*.
2. Drag a Loop block onto the Sequence Beam. This makes the program repeat until you stop it. (The Loop block is the second block from the bottom on the Common Palette.) By default the Loop block is configured with the Control item set to **Forever**, so you don't need to make any changes.

Figure 5-4 shows the program at this point, and Figure 5-5 shows the Configuration Panel for the Loop block.

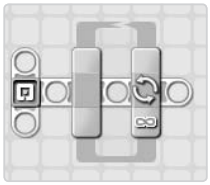


Figure 5-4: Just the Loop block

Figure 5-5: Looping forever

Now you need to add a Move block to the loop to make the robot move forward. Set the Duration item on the Move block to **Unlimited** to make the TriBot keep moving until it bumps into something. (You can keep the default values for the rest of the Move block settings.)

3. Drag a Move block from the Common Palette onto the Sequence Beam inside the Loop block. The Loop block will expand to make room for the Move block.
4. Set the Duration item to **Unlimited**.

Figure 5-6 shows the program so far with the Move block added to the Loop. Figure 5-7 shows the Configuration Panel for the Move block.

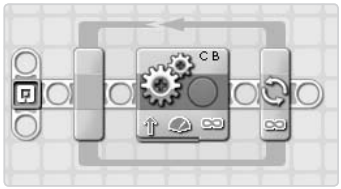


Figure 5-6: With the Move block added to the Loop

Figure 5-7: Configuration for the first Move block

detecting an obstacle

The next part of the program uses the Touch Sensor to tell when the TriBot runs into something, at which point the program stops the motors. A Wait Touch block waits until the Touch Sensor is pressed, at which point you can use a Move block to stop the motors.

5. Drag a Wait Touch block into the Loop block to the right of the Move block. By default the Action item is set to **Pressed**, so keep that setting.
6. Drag another Move block into the Loop block. Set the Direction item to **Stop**.

Figure 5-8, Figure 5-9, and Figure 5-10 show the program at this point and the Configuration Panels for the Wait and Move blocks.

backing up and turning around

The next section of the program makes the TriBot back up a little and turn in a different direction. The robot needs to back up first so it has enough room to turn around. This section uses two Move blocks: one to back up and one to make the robot turn around. The Duration settings for the blocks don't need to be set to any exact value; the robot simply needs to back up enough so that it doesn't hit the object it bumped into while it's turning around.

7. Add another Move block into the Loop block. Click the downward-pointing arrow for the Direction item to make the TriBot back up. Set the Duration item to **300 degrees**.
8. Add another Move block to the Loop. Drag the Steering slider all the way to one side to make the TriBot spin.
9. Set the Duration item to **350 degrees**. You can experiment with different values to see how turning more or less affects the program. (I find it useful to turn at least a quarter turn so that the TriBot doesn't take several tries to move away from a wall.)

Figure 5-11, Figure 5-12, and Figure 5-13 show the final program and the Configuration Panels for the two new Move blocks.

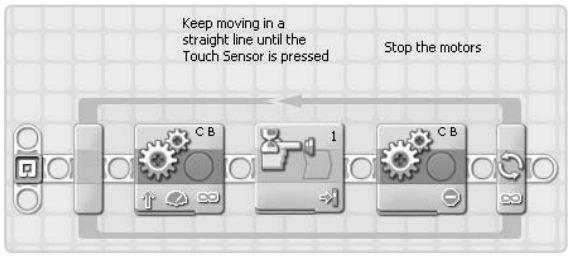


Figure 5-8: Waiting until the Touch Sensor is pressed and then stopping

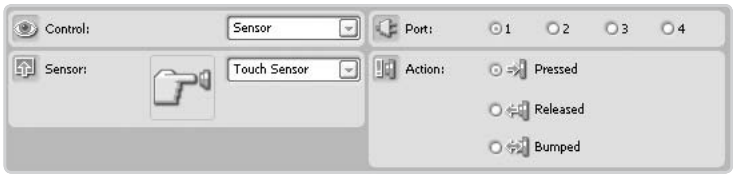


Figure 5-9: Waiting for the Touch Sensor to be pressed

Figure 5-10: Stopping the motors

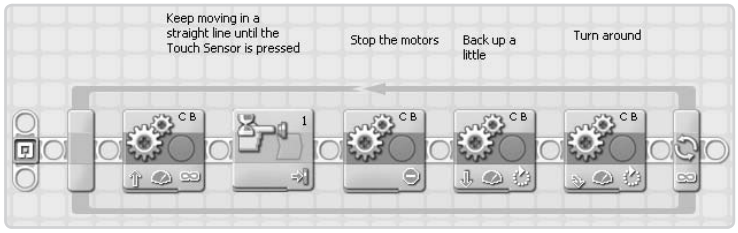


Figure 5-11: BumperBot version 1 complete

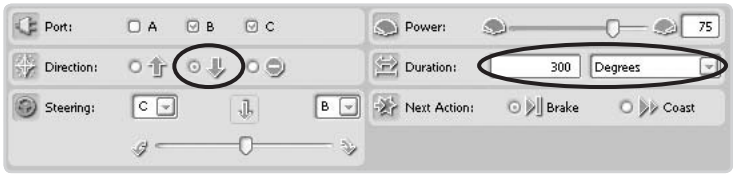


Figure 5-12: Backing away from the obstacle

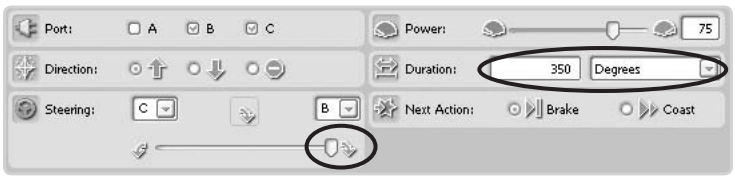


Figure 5-13: Turning around

testing

Now that you've completed the program, it's time to download it to your NXT and test it. When your program runs, the TriBot should move forward in a straight line until it runs into something, at which point it should back away, turn around, and start again. The TriBot should keep running the program until you stop it by pressing the Exit button on the NXT. Experiment with the durations for the last two Move blocks to change how far the robot backs up and turns to find a combination that works well for your test area.

the sound sensor

The Sound Sensor (shown in Figure 5-14) lets your robot respond to sounds, at least in a very simple way. This sensor is included in the original NXT retail kit and the education set. The NXT 2.0 retail kit includes a second Touch Sensor instead of a Sound Sensor.

When working with any sensor, it's important to understand exactly what the sensor is measuring. When we hear a sound, we can discern many characteristics including differences in loudness and pitch. We can even identify different musical instruments by the different sounds they make. The Sound Sensor, however, is much less sophisticated than our ears; it measures only the loudness of a sound.

When you use the Sound Sensor, it will report the loudness of a sound as a number between 0 and 100, with 0 being the softest sound and 100 being the loudest. You can use this loudness measurement as a trigger for a Wait, Loop, or Switch block, allowing your robot to listen for a loud sound or for a room to be very quiet.

When using the Sound Sensor, you set the trigger by comparing the value from the sensor with a target value you supply. To demonstrate how this works, you'll change the BumperBot program so that it waits for you to clap before it starts.

configuration panel

Figure 5-15 shows the Configuration Panel for a Wait block using the Sound Sensor. This panel has the same three sections you saw when using the Touch Sensor, including a Feedback Box on the left, the Wait settings in the middle, and the Sound Sensor settings on the right. All the NXT-G blocks that use sensors have a consistent layout, which is one of the things that makes the NXT-G language so easy to use.

By now you should be familiar with the Port item. The default setting is port 2 for the Sound Sensor. The TriBot instructions had you plug each sensor into its default port, so you won't have to change this setting for any of the sensor blocks.

NOTE **Whenever possible, you should use the default port when attaching the sensors to the NXT. Obviously, you can't do this if your design uses two of the same sensor. For a robot using two Touch Sensors, only one can be plugged into port 1, and the other will need to use one of the remaining ports.**

It's easy to choose an incorrect Port setting, and even though your program will not work, the cause of the problem may not be obvious. The Port setting is the first thing I check whenever it seems that a sensor is not working properly.

setting the trigger value

The Until section of the Configuration Panel is where you set the trigger for the Wait block; you can do this by using the slider or by entering the value directly in the box. You can set the block to trigger when the sensor hears a sound louder than the target by setting the comparison to > (greater than) or clicking the round button on the right end of the slider. To set the block to trigger when the sound level is lower than the target, select the < (less than) comparison, or click the button on the left end of the slider.

Figure 5-14: The Sound Sensor

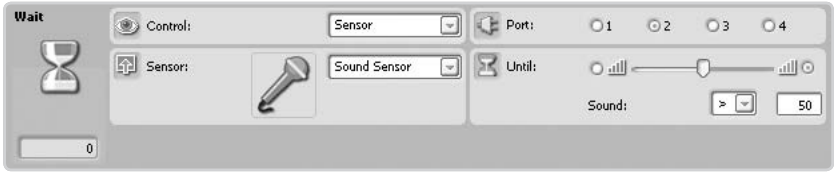


Figure 5-15: Wait block using the Sound Sensor

In the previous chapter, you needed to determine what value to use for the Duration setting of some of the Move blocks. For this program, you need to determine the target value to use to identify a clap. You can use the Feedback Box to find a reasonable value to use; just make sure the NXT is turned on, select the Wait block, and watch the Feedback Box while you clap.

You can also use the NXT's View menu to observe the value from the Sound Sensor. Choose View ▸ Sound dB, and then select 2 for the Port setting. The value displayed on the NXT using this method updates faster than the Feedback Box in the IDE, making it more convenient for short sounds like a clap.

NOTE **You should keep your hands about the same distance away from the robot as they will be when you run your program. If you use the value that shows up in the Feedback Box when you clap an inch away from the sensor, it may not work when you are standing and the TriBot is on the floor. You should try to duplicate the conditions that your robot will need to perform under whenever you run tests like this.**

When I clap, I get a reading of about 70 percent. When I set the trigger on the Wait block, I'll use 60 just to be sure that I don't miss the sound.

dB AND dBA

The NXT's View menu has two options that work with the Sound Sensor: dB and dBA. The loudness of a sound is measured in decibels, abbreviated as dB. To the human ear, the perceived loudness of a sound depends on the pitch. The A-weighted decibel value (dBA) is adjusted so that it matches the way a human ear perceives sound, which makes the dBA value slightly different from the dB value. The sound level used by the Wait, Loop, and Switch blocks is the dB value. You can safely ignore the dBA value; it's useful for some experiments, but anyone needing to use dBA instead of dB will already know the difference.

BumperBot with sound

In this section, you'll create the BumperBotWithSound program by modifying the BumperBot program so that the TriBot waits for you to clap before it starts moving.

To make the program wait for you to clap, you need to add a Wait Sound block before the Loop block. When you drag the new block to the left of the Loop block, the Loop block will shift to the right to make room, as shown in Figure 5-16. It takes some practice to get used to the way the IDE shifts the program around as you are dragging a block. Be sure not to move the block too fast and to pause for a second before dropping the block to make sure you have it in the right spot.

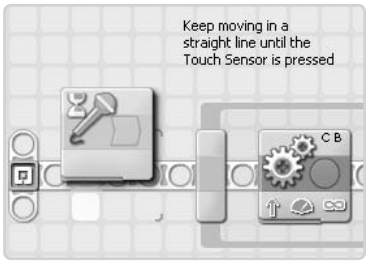


Figure 5-16: Inserting the Wait Sound block

Even after you have been doing this for a while, you'll still occasionally drop a block in the wrong place. When this happens, just select Edit ▸ Undo or press CTRL-Z and try again.

Here are the steps for modifying the program:

1. Open the BumperBot program.
2. Select **File ▸ Save As** and save the program as *BumperBotWithSound*.
3. Drag a Wait Sound block onto the Sequence Beam, between the start of the Sequence Beam and the Loop block.
4. The only setting you need to change is the target value. A target of 60 works well for me, but you may need a lower value if you clap more softly. A value too low can cause the robot to start moving before you clap if you are not in a very quiet place.

Figure 5-17 and Figure 5-18 show the completed program and the Configuration Panel for the Wait block. Download and run the program, and the TriBot should patiently wait for you to clap before it starts moving.

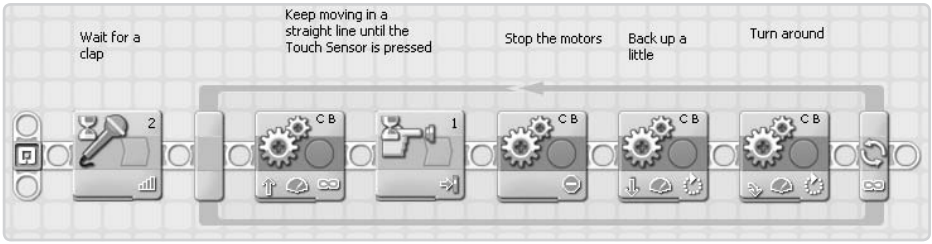


Figure 5-17: The BumperBotWithSound program

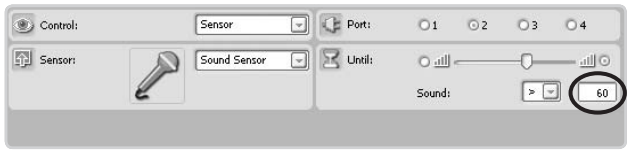


Figure 5-18: Waiting for a clap

the light and color sensors

The Light Sensor (shown in Figure 5-19) measures the brightness of light shining into the front of the sensor. This sensor is useful when you want your robot to react to different light levels in a room or to seek out a light source in a maze or obstacle course. The Light Sensor can distinguish between shades of gray when it's pointed at an object or the floor, which makes it useful for identifying objects or following a line.

The Light Sensor is included in the original NXT retail kit and the education set. The NXT 2.0 retail kit includes a Color Sensor (shown in Figure 5-20) instead of a Light Sensor. The Color Sensor has all the functionality of the Light Sensor, with the added ability to more accurately (and easily) distinguish colors.

Figure 5-19: The Light Sensor

Figure 5-20: The Color Sensor

light sensor configuration panel

Figure 5-21 shows the Configuration Panel for a Wait block using the Light Sensor. The settings for the Light Sensor are similar to those for the Sound Sensor. The Ports and Until settings work the same way for both sensors, except that the default Port setting for the Light Sensor is 3. The brightness value from the sensor will be between 0 and 100, where the darkest value is 0 and the brightest is 100.

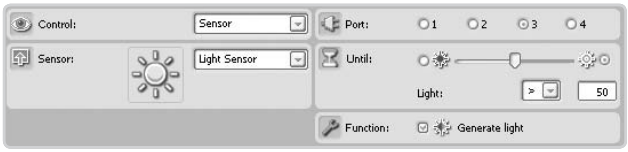


Figure 5-21: Wait block using the Light Sensor

The Light Sensor's Configuration Panel has one additional setting: Generate Light. This setting controls the small light on the front of the sensor. You can use the Light Sensor to measure either ambient or reflected light. *Ambient light* is the light available around the robot, in other words, light that is not created by the robot. When the Generate Light box is not selected, the sensor measures ambient light. This is useful for a robot that responds when a light in a room is turned on or one that searches for a light source.

Select the Generate Light option to measure reflected light. This will turn on the small light on the front of the sensor, and the sensor will measure *reflected light*, which is the amount of light bounced back from an object close to it. This is useful for following a line or identifying objects. Once the light is turned on, it will stay on until you run a block with the Generate Light box deselected.

When measuring reflected light, it's important to place the sensor close to the object you're measuring because the sensor can't tell the difference between the light reflected off the object and any ambient light that is allowed to reach the sensor. Placing the object close to the sensor, or blocking the ambient light some other way, will give you more accurate results.

using the color sensor as a light sensor

The Color Sensor can either measure the intensity of light (like the Light Sensor) or detect the color of an object. To use the Color Sensor in place of a Light Sensor, select Light Sensor for the Action setting on the Configuration Panel (shown in Figure 5-22).

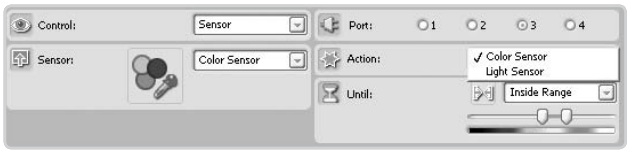


Figure 5-22: Using the Color Sensor as a Light Sensor

Figure 5-23 shows the Configuration Panel for a Wait block using the Color Sensor with the Action item set to Light Sensor. The Until setting works like the Light Sensor counterpart. The Light option is similar to the Generate Light option, except that the Color Sensor contains three colored lamps (red, green, and blue) so you can select the color to use when measuring reflected light.

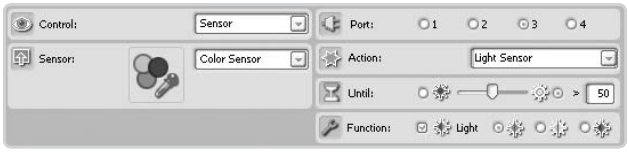


Figure 5-23: Control Panel with the Action item set to Light Sensor

All the programs that use the Light Sensor will also work with the Color Sensor. Just set the Action item to Light Sensor, and use the red lamp when measuring reflected light (to match the Light Sensor's lamp). You'll also need to use different trigger values because the two sensors don't give exactly the same values.

the RedOrBlue program

In this section, you'll build the RedOrBlue program, which uses the Light Sensor (or Color Sensor) to identify the blue and red balls that come with NXT (either the large ones that come with the original NXT retail kit and education set or the small ones that come with the NXT 2.0 retail kit). The Light Sensor measures the intensity of light; it does not measure the color directly. However, the amount of light an object reflects to the sensor depends on the color, and the difference between the Light Sensor readings for the blue and red balls is large enough for you to use it to distinguish between the two. When you run the program, the robot will say either "Red" or "Blue," depending on which ball is placed in front of the sensor.

determining red and blue values

Before you start writing the program, you need to determine the Light Sensor reading for each color. Once again, you can use the Feedback Box on the Wait block or the View menu on the NXT.

The NXT's View menu has two selections for working with the Light Sensor: Reflected light and Ambient light. The small light on the sensor will turn on if you select Reflected light and turn off if you select Ambient light. For this program, select View ▸ Reflected light; the sensor's light should come on while it's displaying the value from the sensor and then turn off when you are done.

The Ambient light and Reflected light options on the NXT's View menu work only with the Light Sensor, not the Color Sensor. Although the Color Sensor can measure both reflected and ambient light, there isn't an option for this on the NXT's View menu. If you're using the Color Sensor, use the Feedback Box on the Wait block to read the values for the blue and red balls.

Now hold each ball, one at a time, very close to the sensor, and read the value from the sensor. I get 55 for the red ball and 27 for the blue ball using the Light Sensor. Using the Color Sensor, I get 43 and 16. If you used the Feedback Box to do this test, you may have noticed that the sensor's light turned on. That little light will stay on until you deselect the Generate Light box (or turn off the NXT), even if you exit the MINDSTORMS environment.

For this program, you'll make what is known as a *simplifying assumption*. This means that instead of solving a hard problem, such as detecting many different colors, you'll solve a simpler problem. For now, assume that all the objects you test are either red or blue. You'll enhance the program later; for now, distinguishing red from blue is good enough.

the switch block

You already know how to use the Sound block to play a sound, so making the robot say "Red" or "Blue" shouldn't be a problem. Of course, to make that pronouncement, the program first needs to make a decision based on the Light Sensor so that the correct sound is played.

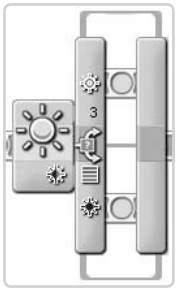


Figure 5-24: The Switch block

In NXT-G, the Switch block (shown in Figure 5-24) is used for making decisions. For now, you'll just use this block in your program. I'll cover its use in detail in the next chapter.

The Configuration Panel for the Switch block, shown in Figure 5-25, has the same layout as the Wait and Loop blocks. You use the items in the center section of this panel to let the block know you want to make a decision based on the Light Sensor. When you select the Light Sensor from the Sensor list, the right side of the Configuration Panel should show the settings for the sensor.

Figure 5-25: Switch block using the Light Sensor

The Compare section is where you tell the block what decision to make, by setting the target value. This section is labeled *Compare* for the Switch block and *Until* for the Wait block, because each block uses the target in a unique way. Even though the labels are different, the target value is set in the same way for each block.

The Switch block looks a little like the Loop block, except that there are two Sequence Beams on the inside instead of one. For example, Figure 5-26 shows the first version of the RedOrBlue program. The Switch block will run the blocks you place on the upper Sequence Beam if the condition you set in the Compare section is met. If the condition is not met, the blocks on the lower Sequence Beam will be used.

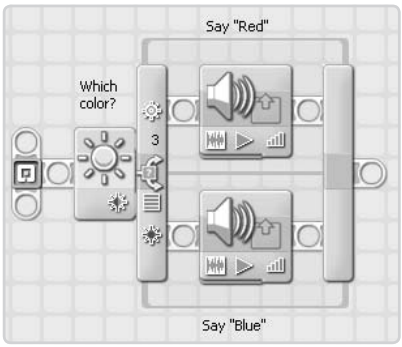


Figure 5-26: First version of RedOrBlue

The values I got from the Light Sensor were 55 for red and 27 for blue, so I'll set the target of the Switch block to 42, which is in the middle, and leave the comparison set to > (greater than). (Set the target to 30 for the Color Sensor.) By using the value midway between 55 and 27, I'm making sure that any object that reads closer to 55 will be identified as red and that, conversely, any object that reads closer to 27 will be identified as blue. When the Switch block is run, the upper Sequence Beam will be used for a red object, and the lower Sequence Beam will be used for a blue object. (Remember, all objects are either red or blue.) Follow these steps to get started:

1. Create a new program called *RedOrBlue*.
2. Drag a Switch block onto the Sequence Beam. The Switch block is at the bottom of the Common Palette.
3. If you're using the Light Sensor, select **Light Sensor** from the Sensor list, and set the Compare value to **42**. The Configuration Panel should look like this:

4. If you're using the Color Sensor, select **Color Sensor** from the Sensor list, set the Action value to **Light Sensor**, and set the Compare value to **30**. The Configuration Panel should look like this:

The next step is to drag a Sound block onto the upper Sequence Beam of the Switch block. This is the block that will be run when the Light Sensor detects a red object, so set the block to say **Red**. The Switch block should expand to the right as you drag the block over it; just go slowly and pause before dropping the Sound block. The only configuration change to make is to select **Red** from the Sound File list.

5. Drag a Sound block onto the upper Sequence Beam inside the Switch block.
6. Select **Red** from the Sound File list. The Configuration Panel should look like this:

To finish this version of the program, drag a Sound block onto the lower Sequence Beam, and set its Sound File setting to **Blue**. Adding this second block will be a little easier than the first one because the Switch block is already expanded.

7. Drag another Sound block onto the lower Sequence Beam inside the Switch block.
8. Select **Blue** from the Sound File list for this block. The Configuration Panel should look like this:

At this point, your program should look like Figure 5-26 (except that the icon on the Switch block will differ if you're using the Color Sensor). Before running the program, be sure that the ball is in front of the Light Sensor. When you run the program, it should correctly identify the red or blue ball. The program should end after saying either "Red" or "Blue," so run it again to test the other color.

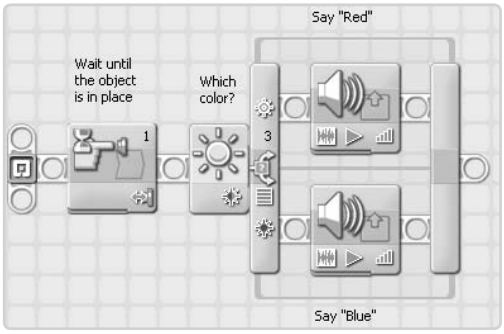
improving the program

The RedOrBlue program works in its current form, but it's a little inconvenient to use. You can improve it by adding a way to tell the robot when the ball is in place. You can also have the program keep running and identifying objects until you stop it.

using the touch sensor

The first change uses the Touch Sensor to let the program know when you are ready to use the Light Sensor. Many programs use the Touch Sensor like this to let the user control the program.

1. Drag a Wait Touch block onto the Sequence Beam to the left of the Switch block. The program should wait until the Touch Sensor is bumped, so this block has to be placed before the Switch block. The program should now look like this:



2. Set the Action item on the Wait Touch block to **Bumped**. The Configuration Panel should look like this:

adding a loop

To make the program keep running, you'll now add a Loop block, and then move the existing blocks inside the Loop.

3. Add a Loop block to the end of the program.

You can move the Switch block and the Wait block into the Loop block separately or select them both and move them as a group. One way to select both blocks is by dragging a selection rectangle around the two blocks, as shown in Figure 5-27. To do so, hold down the left mouse button at one corner of the rectangle and drag the mouse to the other corner. When you release the mouse button, any block within or touching the rectangle will be selected. (Just make sure that you don't include the Loop block.)

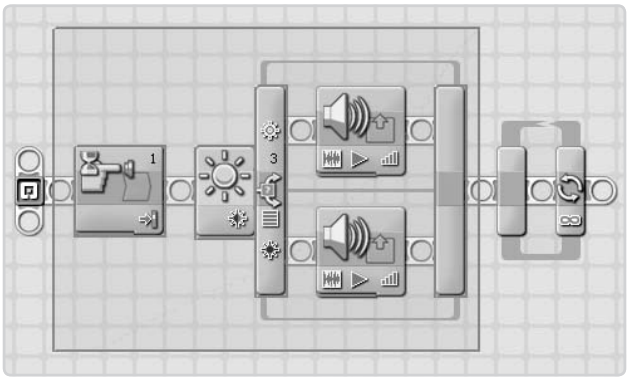


Figure 5-27: Selecting both blocks

Another way to select the two blocks is by clicking the Wait block to select it and then holding down the SHIFT key while clicking the Switch block. Holding the SHIFT key down when clicking a block will add it to the group of previously selected blocks. You can tell which blocks are selected by the light blue outline that appears around any selected block.

4. Select the Wait Touch and Switch blocks, and drag them into the Loop block.

Figure 5-28 shows the new version of the program. Now when you run the program, it will wait for you to hit the bumper and then say either "Red" or "Blue." The program then goes back to waiting again. Use the dark gray button on the NXT to end the program.

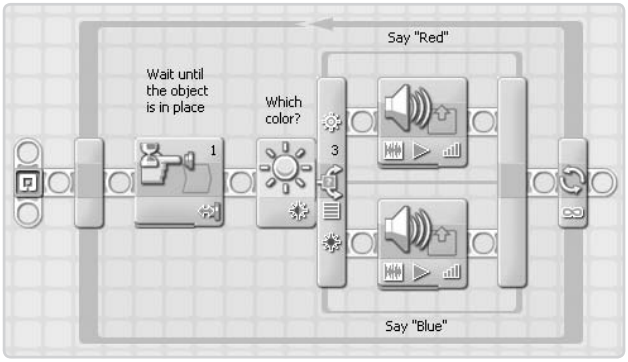


Figure 5-28: The improved version of RedOrBlue

using color sensor mode

The RedOrBlue program described earlier works equally well with either the Light Sensor or the Color Sensor. The Color Sensor can operate in two ways: It can measure the brightness of light (Light Sensor mode), or it can determine the color of an object (Color Sensor mode). When using the Color Sensor, you can improve the program by using the sensor's ability to detect colors directly. Using Color Sensor mode will make the program simpler and more accurate than using Light Sensor mode.

To set the mode for the Color Sensor, use the Action item. Figure 5-29 shows the Configuration Panel for the Switch block with the sensor's Action item set to Color Sensor (instead of Light Sensor). In this mode, the Compare section contains a two-sided slider for selecting the target color. By adjusting the two ends of the slider, you can select a single color or a range of adjacent colors. You can also set the trigger to be inside or outside the range of colors selected.

In Color Sensor mode, the Feedback Box will display a number from 1 to 6 indicating the color detected, according to the mapping shown in Table 5-1.

table 5-1: feedback box values

version	sensors
1	Black
2	Blue
3	Green
4	Yellow
5	Red
6	White

Follow these steps to change the RedOrBlue program to use Color Sensor mode:

1. Open the RedOrBlue program, and save it as *RedOrBlueColorMode*.
2. Select the Switch block.
3. Change the Action setting to **Color Sensor**.

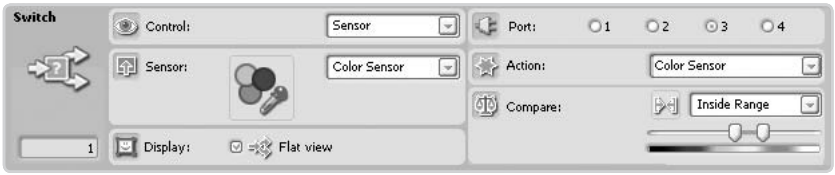
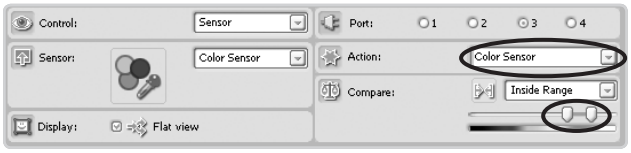


Figure 5-29: Color Sensor mode

4. Move the sliders in the Compare section to select just the red part of the bar. The Configuration Panel should look like this:



When you run this version of the program, it should recognize the red ball as well as the original version, and it will do a much better job of correctly identifying other red objects. (Unfortunately, it will still call all nonred objects blue, which is a problem that you'll fix in Chapter 14.)

the ultrasonic sensor

The Ultrasonic Sensor (shown in Figure 5-30) measures the distance between the sensor and an object. The sensor sends out high-frequency sound waves and measures the time it takes for them to be reflected, which allows it to determine the object's distance from the sensor. (Bats use sonar in the same way.)

Figure 5-30: The Ultrasonic Sensor

An object's shape and texture greatly affect how well it will be detected by the Ultrasonic Sensor; some surfaces reflect sound waves better than others and therefore are easier to detect. Flat, hard surfaces are the easiest to detect because they reflect most of the sound waves back toward the sensor. Curved surfaces reflect some sound waves back to the sensor, but some waves go off in different directions. Soft objects tend to absorb sound waves instead of reflecting them. Consequently, the sensor will be able to detect hard, flat objects more accurately and at a greater distance than soft, round ones.

configuration panel

Figure 5-31 shows the Configuration Panel for a Wait block that uses the Ultrasonic Sensor. The layout is similar to the Configuration Panels for the Sound and Light Sensors. For this sensor, the Until section (where you set the trigger) uses distance instead of the Light or Sound level. The bottom section, labeled Show, lets you select the unit to use (either inches or centimeters).

The Ultrasonic Sensor can measure the distance to an object that is closer than 100 inches or 250 cm away from the sensor. The largest value you can set for the trigger is 250 because that's the sensor's maximum range when using centimeters. When using inches, the largest value the sensor will read is 100, so you shouldn't set the trigger to a higher value. When setting the trigger, be sure to set the units first and then the distance, or the distance will change to match the new units of the new selection.

The value displayed in the Feedback Box will use the units that you select. The Feedback Box will show 100 inches or 255 cm when no object is detected.

The Ultrasonic inch and Ultrasonic cm options on the NXT's View menu display the distance to an object using the appropriate units. The NXT will display `??????` when no object is detected.

door chime

The next program you'll create is a simple door chime: You load it onto your TriBot, and place the TriBot at a doorway with the Ultrasonic Sensor facing the opening. The TriBot will chime when someone walks through the door and then start over in preparation for the next person.

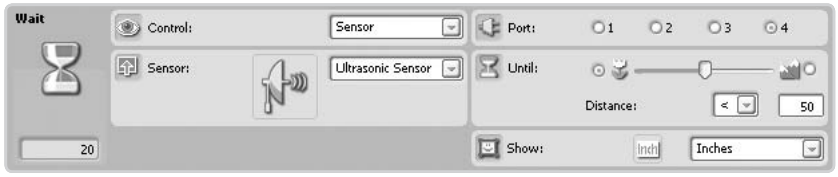


Figure 5-31: Wait block using the Ultrasonic Sensor

When writing a program that loops around like this, it doesn't matter whether you start with the Loop block and build the program inside the loop or wait until the program works once and then add the Loop block. For this example, you'll start with the Loop block and build the program inside it:

1. Create a program with the name *DoorChime*.
2. Add a Loop block to the Sequence Beam. For this program, the loop should continue until the program is stopped, so you don't need to change any of the settings.

detecting a person

You can use a Wait block with the Ultrasonic Sensor to detect a person walking past the robot. On the Common Palette, the tool tip for this type of Wait block says "Distance."

Once again you need to determine the value to use for the trigger. To do so, use the Feedback Box or the NXT's View menu to see what the sensor reads when you place it next to the doorway, or you can measure the doorway to find a value that should be accurate enough. The doorway I'm using is 32 inches wide. If I place the TriBot next to the opening, when a person walks through the door, they'll be closer than 32 inches from the robot. Therefore, I'll use 32 inches in the following instructions. The Wait block should trigger when the sensor detects something closer than your target value, so leave the comparison set to less than (<).

3. Add a Wait Distance block inside the Loop block.
4. Set the Until setting to your trigger value.

Figure 5-32 and Figure 5-33 show the program so far and the Configuration Panel of the Wait block.

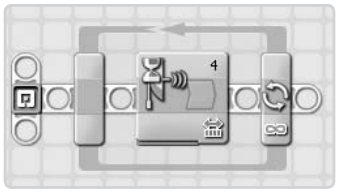


Figure 5-32: Waiting for a person

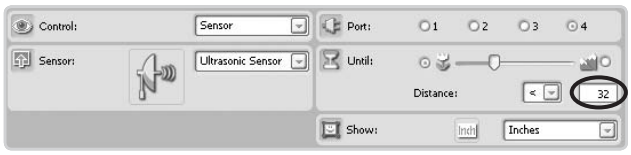


Figure 5-33: Configuration Panel for the Wait block

playing a chime

Once the program detects someone walking by, it should play a chime. You can use two Sound blocks with the Action item set to Tone to play any two notes you like. You can experiment with different combinations or even add more notes if you want.

5. Add a Sound block inside the Loop.
6. Set the Action item to **Tone**. Select a note by clicking the piano keyboard on the right side of the Configuration Panel.
7. Add another Sound block inside the Loop.
8. Set the Action item to **Tone** and select a different note.

Figure 5-34, Figure 5-35, and Figure 5-36 show the program and Configuration Panels for the two Sound blocks. Download and run the program to see whether it works. Experiment with different distances to find the trigger value that works best for you.

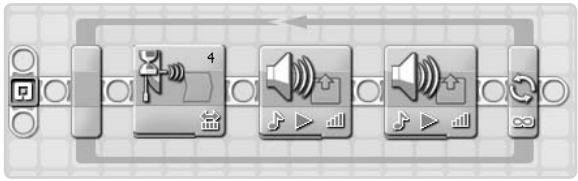


Figure 5-34: With the chime added

Figure 5-35: Configuration of the first Sound block

Figure 5-36: Configuration of the second Sound block

stopping the chime

When a person walks by the doorway, the TriBot should chime. But what if they stop and stand in the doorway? The Ultrasonic Sensor will keep reading a value that is less than the trigger value, and the TriBot will just keep chiming until they move along.

To solve this problem, add another Wait block after the two Sound blocks. This block will pause the program until the person moves beyond the doorway by waiting for the Ultrasonic Sensor to give a reading that is greater than the original trigger value (32 inches for my doorway).

9. Drag a Wait Distance block inside the Loop to the right of the Sound blocks.
10. Change the Until setting to be greater than the trigger value you used for the first Wait block. Be sure to change both the value and the comparison.

Figure 5-37 and Figure 5-38 show the Configuration Panel for the new block and the completed program. Test this version of the program to see whether it behaves rationally as you move through the doorway. Experiment with different distances, and test to see how the program behaves if more than one person moves through the doorway.

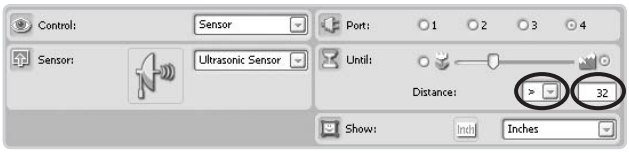


Figure 5-37: Waiting for the person to move away

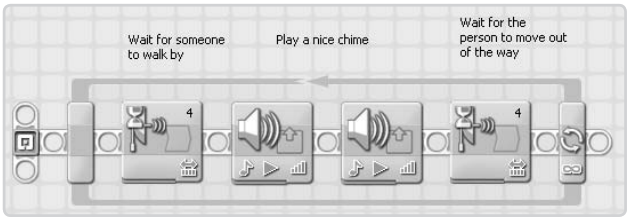


Figure 5-38: The completed DoorChime program

the rotation sensor

Each NXT motor has a Rotation Sensor built into it that measures how far the motor has turned. You can use this sensor to control how far the robot moves as an alternative to the Move block's Duration setting. This gives you an easy way to make the program perform certain actions while the robot is moving. For example, you'll change the BumperBot program to make the TriBot beep while backing up.

While your program is running, the Rotation Sensor tracks how far the motor has moved. You can read this value at any time, the same way you read values from the other sensors. You can also reset the Rotation Sensor value to 0, allowing you to measure how far the motor has moved between any two points in the program. (In most cases, you'll use two blocks when using the Rotation Sensor, one to reset the value and one to read it.)

configuration panel

Figure 5-39 shows the Configuration Panel for a Loop block using the Rotation Sensor. The Port setting is used to select the motor to use.

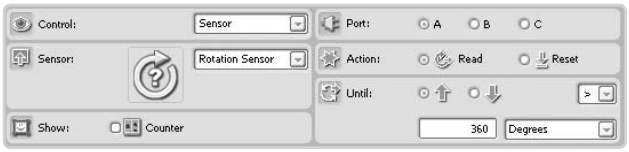


Figure 5-39: Configuration Panel for the Loop block using the Rotation Sensor

The items in the Until section are used to set the trigger value, in either degrees or rotations. The arrows are used to select the direction to which the trigger applies. The Rotation Sensor always reads a positive number (or 0), so you need to specify the direction as well as the distance.

The Action setting determines whether the block should read the rotation value or reset it to 0. When used with a Loop or Switch block, the value is always read first and tested against the trigger value; then the value is set to 0 if the Reset option is selected. The Action item is disabled when a Rotation Sensor is used with the Wait block.

the rotation sensor block

Use the Rotation Sensor block (grouped with the other sensor blocks on the Complete Palette) to reset the Rotation Sensor to 0 outside a Loop or Switch block (as shown in Figure 5-40). When you add this block to your program, it will look like Figure 5-41. You'll use this block to reset the Rotation Sensor here, and you'll revisit it in Chapter 8.

The Configuration Panel (shown in Figure 5-42) has the same settings for controlling the Rotation Sensor as described earlier, without the section for controlling the loop.

Figure 5-40: The Rotation Sensor block on the Complete Palette

Figure 5-41: The Rotation Sensor block

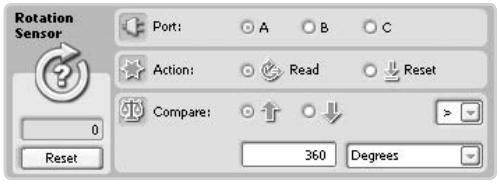


Figure 5-42: Configuration Panel for the Rotation Sensor block

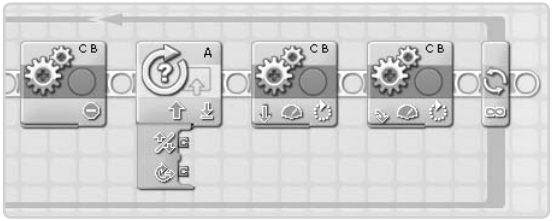
the BumperBot2 program

In the original version of the BumperBot program, a Move block with the Duration item set to 300 degrees was used to make the TriBot back up after it hit something (see Figure 5-43). But what if you want the robot to beep while backing up, like a school bus or large truck?

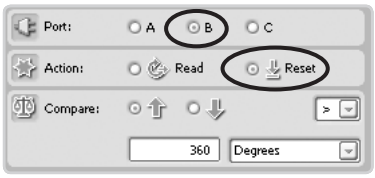
You already know how to use the Sound block to make the robot beep; the real challenge comes in making it beep while moving. One way to do this is to start the robot moving using a Move block with the Duration item set to Unlimited and then use a Loop block to stop the robot after it has moved 300 degrees. Inside the Loop block, you can use a Sound block to make the robot beep.

The following instructions replace the Move block circled in Figure 5-43 with blocks to reset the Rotation Sensor, start the TriBot moving, and then beep until the Rotation Sensor reads 300 degrees.

1. Open the BumperBot program.
2. Save the program as *BumperBot2*.
3. Add a Rotation Sensor block to the left of the Move block highlighted earlier. This part of the program should now look like this:

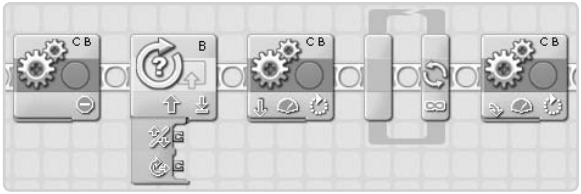


4. Set the Port item to **B** and the Action item to **Reset**. The Configuration Panel should look like this:



5. Change the Duration setting of the Move block highlighted in Figure 5-43 to **Unlimited** and the Power to **25**. This will keep the TriBot moving back slowly until you stop it. The Configuration Panel should now look like this:

6. Add a Loop block to the right of the Move block. This part of the program should now look like this:



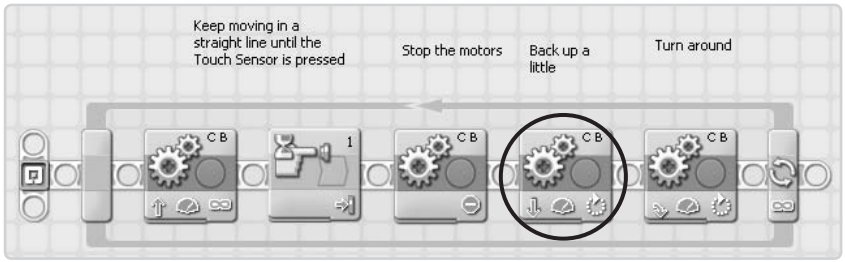
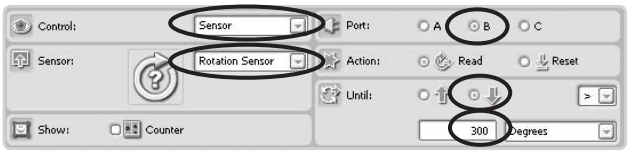


Figure 5-43: The Move block used to back up

7. Set the Loop block to use the Rotation Sensor. Set the Port item to **B**, and set the trigger value to **300 degrees**. Finally, select the downward-pointing arrow in the Until section (because the motor is moving backward). The Configuration Panel should look like this:



8. Drag a Sound block into the Loop block, and set the Action item to **Tone**. The Configuration Panel should look like this:

9. Place a Wait Time block to the right of the Sound block. Set the time to wait to **0.25** seconds. This gives a little pause between the beeps. The Configuration Panel should look like this:

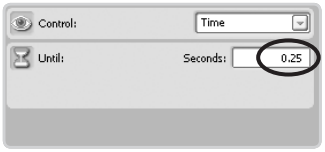


Figure 5-44 shows the changed part of the program. When you run the program, the TriBot should back up slowly while beeping after it bumps into something.

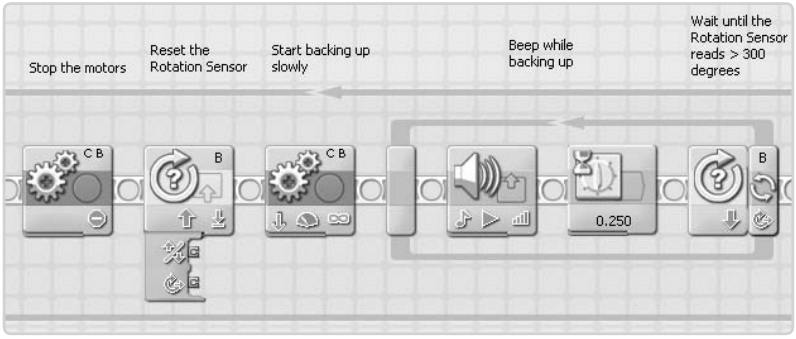


Figure 5-44: The new code used to back up

conclusion

The NXT sensors allow you to build a robot that interacts with the world around it. The different sensors (Light, Color, Touch, Sound, Ultrasonic, and Rotation) let your robot perceive its environment in a several different ways, allowing you to create robots that exhibit a wide variety of interesting behaviors. The programs presented in this chapter show how to use of each of the sensors in combination with Wait, Loop, and Switch blocks to create some interesting NXT-G programs, from a simple door chime to the more complex BumperBot program.

6

program flow

Program flow is all about controlling the order in which the blocks in your program are run. As you can imagine, controlling the order of your blocks is as important as setting their configuration. The parts of a programming language that control the program flow are often called *programming structures*, and NXT-G has three: the Sequence Beam, the Switch block, and the Loop block.

In this chapter, I'll cover the Switch and Loop blocks in depth and show you how to use them effectively, filling in the details I glossed over in the previous chapters. A few other blocks are also used to control how your program runs. You have already seen the Wait block, and I'll cover the Keep Alive and End blocks in this chapter.

NOTE **The information in this chapter assumes you are not using data wires. The Switch and Loop blocks have some features that are used exclusively with data wires, and I'll cover those features in Chapters 9 and 10.**

the sequence beam

You've been using the Sequence Beam in the previous chapters because you can't write a NXT-G program without it. At this point, you should be pretty comfortable placing blocks on the Sequence Beam and knowing how the blocks will run. With just a few exceptions, each block runs until it finishes, and then the next block on the Sequence Beam starts. The program ends when it gets to the end of the Sequence Beam.

The exceptions mentioned earlier are those blocks that can be configured to start an action and then allow the program to continue, such as the Move block with the Duration set to Unlimited and the Sound and Motor blocks with the Wait For Completion box deselected. These blocks let you continue moving the motors or playing a sound while the rest of the program runs. The program will still end when it reaches the end of the Sequence Beam, even if one of these blocks is playing a sound or moving a motor.

I'll discuss the Sequence Beam again in Chapter 17. Until then, all the example programs will use a single Sequence Beam and operate as described earlier.

the switch block

The Switch block (shown in Figure 6-1) lets your program make a decision about which blocks to run. This type of structure is called a *conditional* because the program flow changes based on some condition. The Switch block uses the condition to choose from two or more groups of blocks, giving a program the ability to make

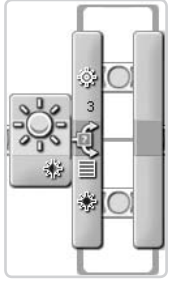


Figure 6-1: The Switch block

a decision and react to the data read by the robot's sensors. For example, the RedOrBlue program uses the reading from the Light Sensor for the test condition to decide which Sound block to run.

configuration panel

Figure 6-2 shows the Configuration Panel for the Switch block used in the RedOrBlue program. The three items on the left, Control, Sensor, and Display, affect the Switch block directly. The configuration items on the right control the selected sensor.

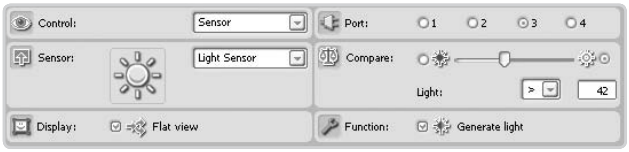


Figure 6-2: The Switch block's Configuration Panel

the control setting

The Control setting determines where the block gets the data it uses to make its decision. The data can be supplied by either a sensor or a value passed into the block using a data wire, so this item has two choices: Sensor and Value. In this chapter, I'll limit the discussion to the Sensor option and cover the Value option in Chapter 9.

the sensor setting

The list of Sensors, shown in Figure 6-3, includes the standard NXT sensors, the Rotation sensor (part of the NXT motor), the buttons on the NXT, a timer, and messages sent between NXTs using Bluetooth.

The list of sensors is slightly different for each version of the MINDSTORMS software. The MINDSTORMS 1.1 Education software includes additional choices for the older RCX sensors, listed as Light* Sensor, Touch* Sensor, Rotation* Sensor, and Temperature* Sensor. The MINDSTORMS 2.0 Education software includes the Temperature Sensor. The list shown in Figure 6-3 is from the MINDSTORMS 2.0 Retail software.

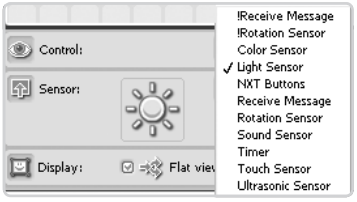


Figure 6-3: The Sensor options

The top two items in the list shown in Figure 6-3, !Receive Message and !Rotation Sensor, appear in both the MINDSTORMS Education and Retail 2.0 releases. These 2.0 versions of the software include small changes to the way the Receive Message and Rotation Sensor choices work. Therefore, the new !Receive Message and !Rotation Sensor choices retain the behavior from the original MINDSTORMS software release to make it easier to reuse programs written using the older software. When writing new programs with the MINDSTORMS Software 2.0, you should use the regular Receive Message and Rotation Sensor choices.

setting the condition

When you select a sensor from the list, the right side of the Configuration Panel will show the settings for that sensor. Each of the sensors has some way to set the condition, also known as the *trigger*. For example, the trigger for the Light Sensor is set in the Compare section, shown in Figure 6-2. Figure 6-4 shows the Switch block's Configuration Panel using the Touch Sensor. In this case, the trigger is set by selecting the desired Action setting, either Pressed, Released, or Bumped.

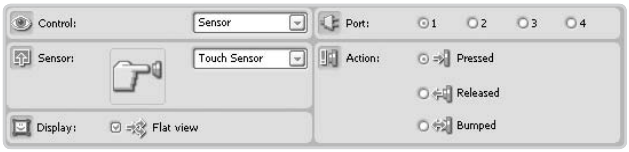


Figure 6-4: Setting the trigger for the Touch Sensor

Inside the Switch block are two Sequence Beams. Your program will run the blocks on one of the Sequence Beams based on the condition. The blocks on the upper Sequence Beam will be used if the condition is true, and the blocks on the lower Sequence Beam will be used if the condition is false. The process for setting the condition will depend on the sensor you're using and the behavior you want.

the display setting

The Display setting controls how the Switch block is displayed in the Work Area. By default the Flat view option is selected, which displays both the upper and lower Sequence Beams. Deselecting the Flat view box will display the block using Tabbed View, which shows only one of the two Sequence Beams. Figure 6-5 shows the Switch block from the RedOrBlue program using Tabbed View. The two tabs at the top of the switch structure are used to select which Sequence Beam to display. The icons used on the tabs depend on the sensor and the condition set. In this case, because the condition is looking for a light level over

a certain value, the icon for a bright light is used for the first tab, and one for a darkened light is used for the second tab.

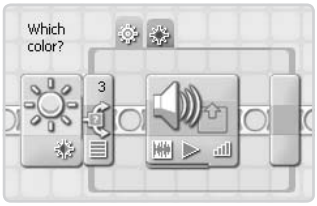


Figure 6-5: Switch block using Tabbed View

the LineFollower program

The next program is a simple line follower. The TriBot uses the Light Sensor to follow a black line on a white background. The idea is to have the TriBot follow the edge of the line by adjusting the steering based on the Light Sensor reading. The discussion and the screenshots refer to the Light Sensor; however, the Color Sensor works just as well.

For this program, you need to remove the Touch Sensor bumper from the front of the TriBot and replace it with the Light Sensor. The Light Sensor should be mounted so that it points downward, as shown in Figure 6-6.

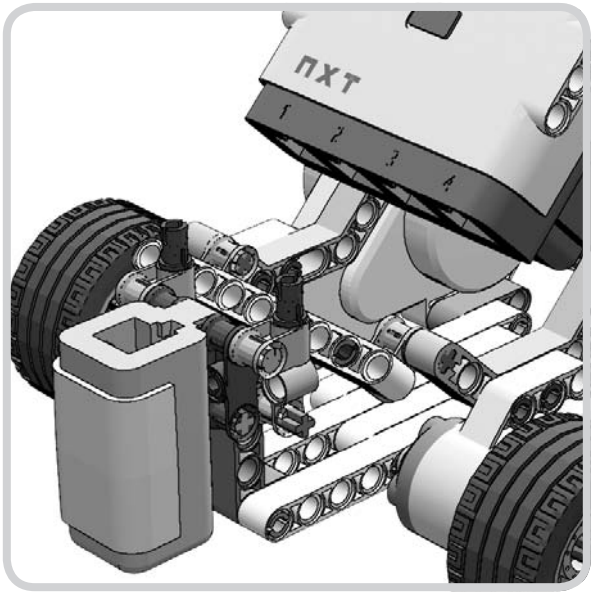


Figure 6-6: Light Sensor position for following a line

To test this program, you'll need a line to follow. Both versions of the NXT retail kit include a test pad that works well with this program, but you can also create your own test pad using black electrical tape on a white poster board.

NOTE **The early versions of this program will work best with a smooth oval. The test pad included in the original NXT retail kit has a few sharp corners at one end that may cause some problems.**

the basic program

Figure 6-7 shows the basic structure of the program, with the blocks all in place but not yet configured. (The blocks will appear a little different after following the instructions given in this section.) The Loop block keeps the program running until you stop it. The Switch block reads the Light Sensor and decides which Move block to run. The Move block on the upper Sequence Beam steers the TriBot to the left, and the one on the lower Sequence Beam steers it to the right.

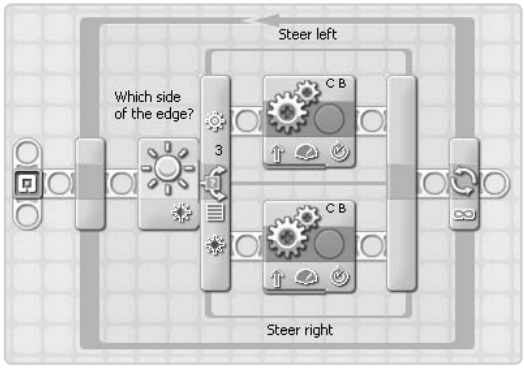


Figure 6-7: LineFollower program structure (before configuring the blocks)

These are the first steps in creating this program:

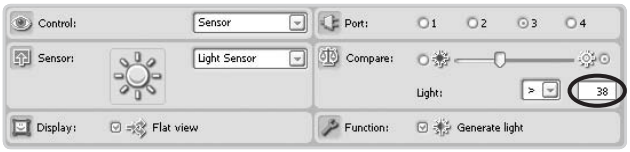
1. Create a new program named *LineFollower*.
2. Add the Loop, Switch, and Move blocks as shown in Figure 6-7.
3. Select the Switch block, and configure it to use the Light Sensor. The Configuration Panel should look like this:

4. If you're using the Color Sensor, select it from the Switch block's sensor list, and set the Action item to **Light Sensor**. The Configuration Panel should now look like this:

selecting the light sensor trigger

Now it's time to determine the trigger value to use for the Switch block. To make the TriBot follow the edge of the line, you'll need to find out what value the Light Sensor reads when it's over the edge of the line. To do so, place the TriBot on the test pad so that the Light Sensor is centered over the line's edge, and use the Feedback Box on the Switch block or the NXT's View menu to read the light level. My Light Sensor reads 38, so I'll use that for the Switch block's trigger value. Your value may be a little different depending on the sensor, the test pad, and the lighting in the room.

5. Set the Switch block's trigger value. The Configuration Panel should look like this:

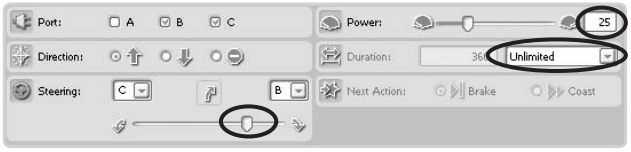


configuring the move blocks

The two Move blocks will have similar settings, except that steering will be in opposite directions. The speed of the motors and the steering setting will significantly affect how well the TriBot follows the line. Begin by setting the steering to about halfway between the middle and the end of the slider with a Power setting of 25:

6. Select the Move block on the upper Sequence Beam.
7. Set the Duration item to **Unlimited** and the Power item to **25**.
8. Move the Steering setting to halfway between the middle and the left end of the slider. The Configuration Panel should look like this:

9. Select the Move block on the lower Sequence Beam.
10. Set the Duration item to **Unlimited** and the Power item to **25**.
11. Move the Steering setting to halfway between the middle and the right end of the slider. The Configuration Panel should look like this:



testing the program

Now download and run the program to see how well it works and how fast you can make the robot move while still staying near the line. You may need to adjust how fast the TriBot moves and how sharply it turns. If you do, just make sure that you make the same changes to both Move blocks.

more than two choices

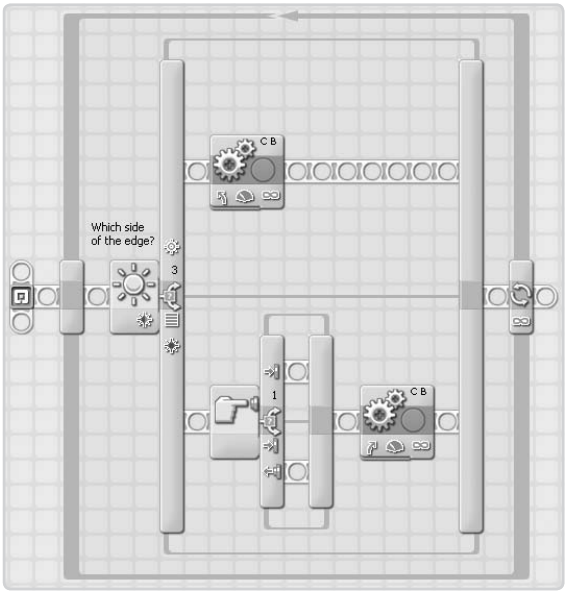
A Switch block can choose only between two sets of blocks based on the reading from the Light Sensor. To choose between three options, you'll need to use two Switch blocks.

The first version of the LineFollower program makes the TriBot wiggle left and right while following a straight line, because the TriBot is constantly adjusting the steering. You can make the motion smoother with three Move blocks: one to steer to the left, one to go straight, and one to steer to the right.

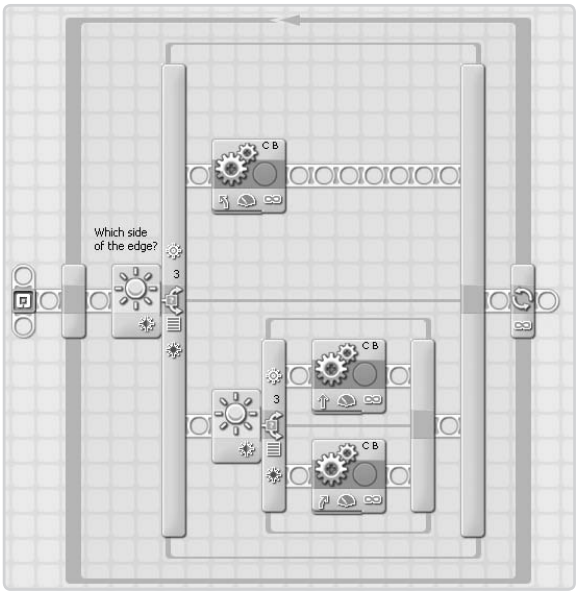
To choose among three Move blocks, you'll need to add another Switch block. The first Switch block will decide if the robot should turn to the left, and the second will decide whether to go straight or turn to the right. This is a pattern that's used often in NXT-G programming and will quickly become familiar.

Start by making the following changes to the Line-Follower program:

1. Place a Switch block on the lower Sequence Beam of the existing Switch block, to the right of the Move block. The program should look like this:



2. Select the Light Sensor from the Switch block's sensor list.
3. Drag the existing Move block (the one that turns to the right) onto the lower Sequence Beam of the new Switch block.
4. Place a new Move block on the upper Sequence Beam of the new Switch block. The program should now look like this:



setting the trigger values

You need to set the trigger values for the two Switch blocks so that the TriBot will go straight when it's on the edge of the line and turn when it moves away from the edge. Start by finding the values that the Light Sensor reads when it's over the center of the line and when it's completely off the line. My readings are 26 and 50.

Based on my readings, I can expect the Light Sensor to read between 26 and 50 and be near 38 when the TriBot is over the line's edge. Therefore, I'll set the trigger values so that the robot drives straight when the reading is between 33 and 43; these are values that are about halfway between the value at the line's edge (38) and the limits I expect to see when it's over the middle of the line or completely off the line (26 and 50). Table 6-1 shows how the program should behave based on the Light Sensor reading.

table 6-1: light sensor ranges and program behavior

light sensor reading	program behavior
0–32	Turn right.
33–42	Go straight.
43–100	Turn left.

Follow these steps to complete this version of the program:

5. Select the outer Switch block (the one from the original program), and then set the trigger value to the upper limit of the range where you want the robot to drive straight (43 using my values).
6. Select the new Switch block, and set the trigger value to the lower limit. To make the Switch block use the upper Sequence Beam when the reading is 33 or greater, you need to set the trigger value to 32.
7. Select the new Move block. Set the Duration item to **Unlimited** and the Power item to **25**.

Figure 6-8, Figure 6-9, and Figure 6-10 show the Configuration Panels for the two Switch blocks and the new Move block. Figure 6-11 shows the completed program.

Figure 6-8: Configuration Panel for the outer Switch block

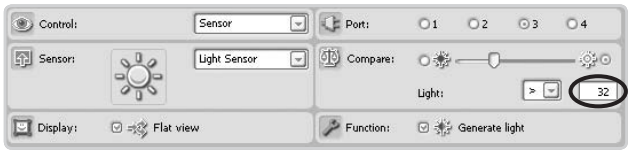


Figure 6-9: Configuration Panel for the inner Switch block

Figure 6-10: Configuration Panel for the new Move block

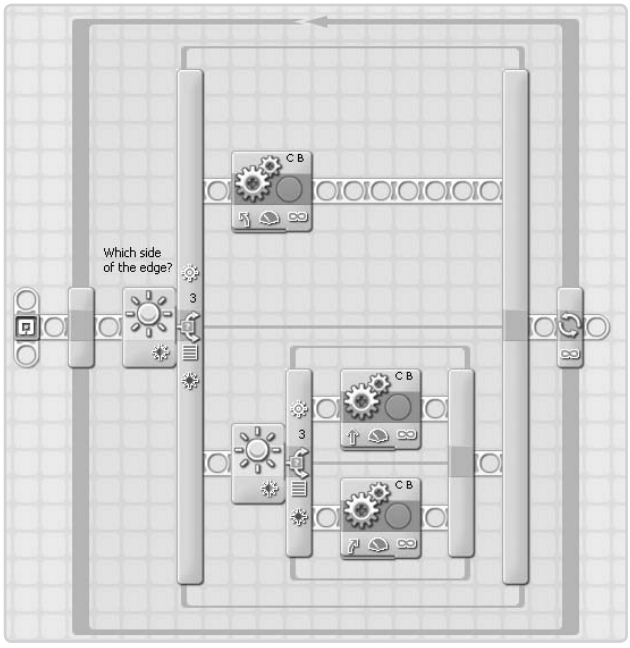


Figure 6-11: The complete LineFollower program

testing the program

Now when you run the program, you should notice that the motion is much smoother when following a straight line. The Power setting for the two Move blocks that steer the robot to the left and right should be the same. However, the setting for the new Move block doesn't need to match the other two, so you can try going faster when moving straight. Experiment with different trigger values and Power and Steering settings to see how fast you can make the TriBot move without veering off the line and getting lost.

using tabbed view

The LineFollower program is an example of *nested* Switch blocks, where one block is nested inside the other. You'll use nested blocks often in programs that need to make complicated decisions.

Nested Switch blocks tend to take up a lot of room on the screen, which can make it difficult to work with the rest of the program. To shrink the size of the Switch blocks, switch to Tabbed View (deselect the Flat view option); the program should take up much less vertical space, as you can see in Figure 6-12. The advantage of Flat view is that you can see the entire structure all at once; the advantage of Tabbed View is that it's easier to work with the rest of the program.

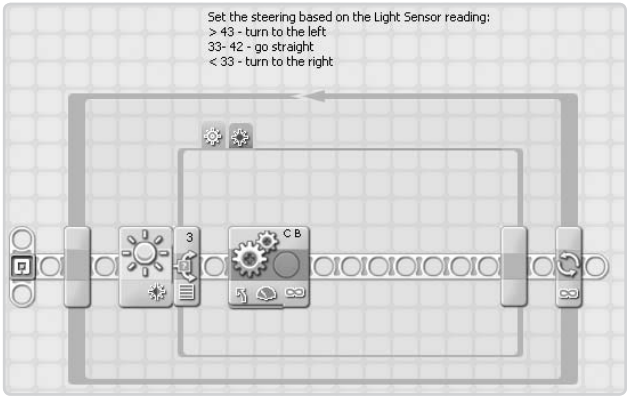


Figure 6-12: The Switch blocks using Tabbed View

comments and tabbed view

Adding comments to a program is an important way to describe how a program works both for your sake and for anyone who might be looking at your program later. You can describe most blocks by simply adding a comment above the block or putting a large comment before a group of related blocks. But in the case of a Switch block using Tabbed View, only some of the blocks are visible at any one time. This means you can see only part of the program logic, especially when using nested Switch blocks. In this case, I prefer to put a large comment above the outermost Switch block to describe the entire section of code. Figure 6-12 shows a reasonable comment for the code to make the robot follow a line.

NOTE **It would be nice if you could add comments close to the blocks within the Switch structure so that you could describe the function of only the visible blocks, but this doesn't work well. In most cases, all the comments**

within the Switch block will be visible, no matter which tab you select. However, if you place a comment in just the right place, it will appear only on a single tab. This seems to confuse the IDE and can make editing the program difficult. The bottom of the block will be cut off by the edge of the Switch block. In addition, you'll be unable to add more blocks within the Switch block. The best approach is to just leave the comments outside the Switch block.

the loop block

The Loop block lets you repeat a group of blocks over and over. The condition you set controls the number of times the loop is repeated.

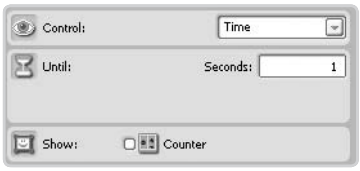
A loop has two parts: the loop body and the loop condition. The *loop body* is the group of blocks within the loop. The *condition* tells the Loop block whether to run the loop body again or exit the loop and let the next block start. The condition is always checked after the loop body runs; therefore, the loop body is always run at least once.

As shown in Figure 6-13, the Loop block's Control item gives you five ways to control how many times the body of the loop is run. Each option is discussed next.

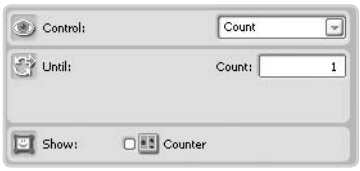
Figure 6-13: Control options for the Loop block

- **Forever.** The Loop will continue to run until the program is stopped. The program can be stopped by pressing the small gray button on the NXT or using the Stop block (described in a moment).
- **Sensor.** A sensor is used to decide when to exit the loop. You set the condition just as with the Switch block, and the loop is exited when the condition is met. For example, the Configuration Panel for a Loop block that repeats until the Touch Sensor is pressed looks like this:

- **Time.** The Loop is run for the specified number of seconds. With this option selected, the Configuration Panel will look like this:



- **Count.** The Loop is run the specified number of times. The Configuration Panel will look like this:



- **Logic.** A value passed into the Loop using a data wire determines whether the Loop should continue. (Chapter 10 discusses using a Loop block with a data wire.)

the keep alive block

The NXT has a built-in sleep setting that turns it off when it's not in use. By default the sleep time is set to either 10 minutes or an hour, depending on the version of the firmware on your NXT. You can use the NXT's Settings menu to set the sleep time or turn off this feature.

Having the NXT go to sleep is useful for saving your batteries; however, you may want to prevent the NXT from turning off while your program is running. That's where the Keep Alive block comes in.

The Keep Alive block is in the Advanced group at the bottom of the Complete Palette (shown in Figure 6-14). The icon for this block may not be immediately obvious. It's a pair of *Z*s (to indicate sleeping) crossed out using a red circle with a line through it. In your program, this block will appear as shown in Figure 6-15.

Figure 6-14: The Keep Alive block on the Complete Palette

Figure 6-15: The Keep Alive block

The Keep Alive block doesn't have any configuration items. When the block is run, it resets the NXT's sleep timer. To keep a program running indefinitely, place this block in a loop so that it is executed more often than the sleep time.

For example, if you were to add this block to the BumperBot2 program (as shown in Figure 6-16), the sleep timer would reset each time around the loop. As long as the TriBot keeps moving around the room, the program will keep running. The NXT *will* go to sleep if the robot gets stuck and the Touch Sensor isn't pressed for too long, which is better than having it spinning its wheels until the batteries wear out.

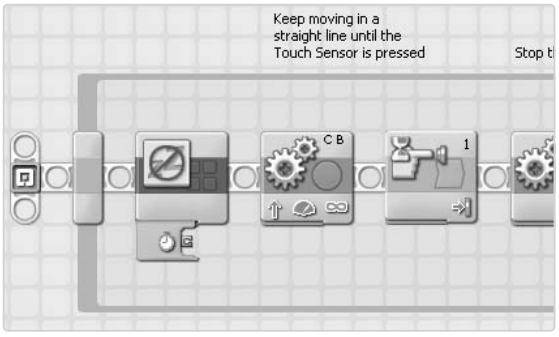


Figure 6-16: Keeping the robot from going to sleep

the stop block

All the programs presented so far have ended either because they reached the end of the Sequence Beam or because you pressed the Exit button on the NXT. The Stop block gives you a way to make a program end itself.

This block is very simple; it has no configuration items, and when it's run, the program ends. You'll find the Stop block on the Complete Palette in the Flow group, as shown in Figure 6-17. Figure 6-18 shows how it will look in your program.

Figure 6-17: The Stop block on the Complete Palette

Figure 6-18: The Stop block

BumperBot3

Now you'll make some changes to the BumperBot2 program to see how to use the Stop block. To use this program, you'll need to have the TriBot assembled in its original configuration, with the Touch Sensor bumper mounted on the front and the Light (or Color) Sensor on the side, as shown here:

The changes you'll make will stop the program when you turn off the lights. Figure 6-19 shows the relevant part of the original program. The code shown here uses a Move block to start the TriBot going forward and then waits for the Touch Sensor to be pressed. The program doesn't do anything else at this point except wait for the Touch Sensor to be pressed. You'll change this to make the program check

the Light Sensor while it's waiting. If the Light Sensor reading is very low, indicating that the lights have been turned off, then the program will stop itself. Monitoring the Light Sensor while waiting for the Touch Sensor to be pressed requires code that is a little more complex than a single Wait Touch block.

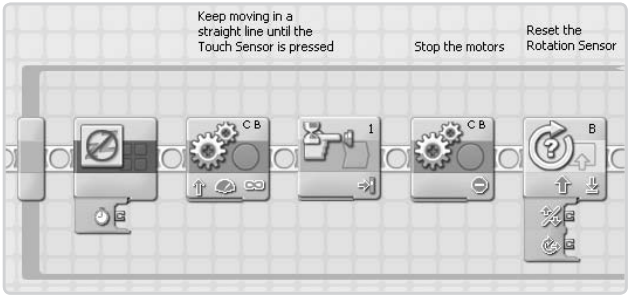


Figure 6-19: BumperBot2 before the changes

The following instructions replace the Wait Touch block with a Loop block that exits when the Touch Sensor is pressed. The Light Sensor is used in the body of the loop. If the Light Sensor doesn't detect enough light, the program will stop the motors, say "Goodbye," and then end. Figure 6-20 shows the code changes because it's easier to understand this example if you can see the final result.

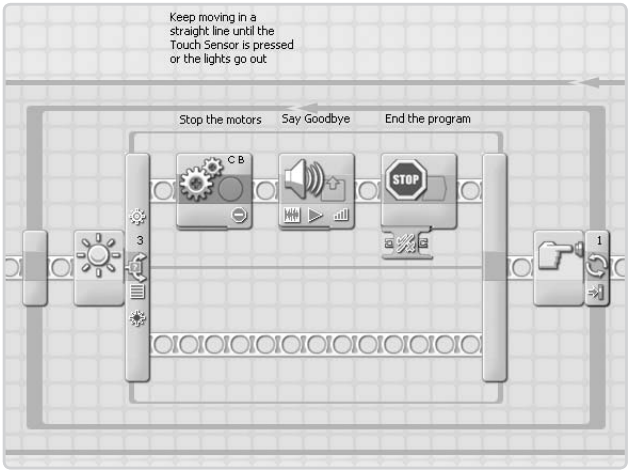
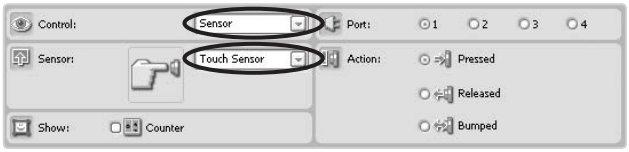


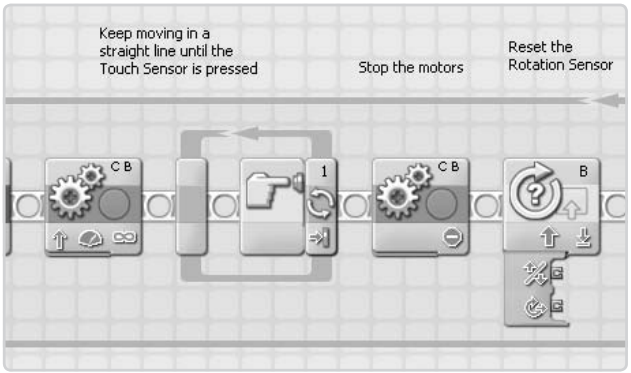
Figure 6-20: The code to replace the Wait Touch block.

Follow these steps to replace the Wait Touch block with the new code:

1. Open the BumperBot2 program.
2. Select **File ▸ Save As** to save the program as *BumperBot3*.
3. Drag a Loop block to the right of the first Move block. Select **Sensor** for the Control item, and select **Touch Sensor** from the list of sensors. You can keep the default values for the Port and Action items. The Configuration Panel should look like this:



4. Delete the Wait Touch block. This part of the program should look like this:



At this point, the program should behave exactly as it did before you made the changes because the Loop block will do the same thing that the Wait block did. The Loop block takes up more space than the Wait block it replaced and isn't as simple to configure. However, it has one big advantage: It gives you a place to put some blocks that will run while the program is waiting.

The next step adds a Switch block inside the Loop to check the Light Sensor:

5. Drag a Switch block into the Loop block.
6. In the Switch block's Configuration Panel, select the **Light Sensor**. Deselect the **Generate Light** box, set the comparison to less than (**<**), and set the trigger value to **10**. You can adjust this value after some testing if it's too high or too low. The Configuration Panel should look like this:

7. If you're using the Color Sensor, the Switch block's Configuration Panel should look like this:

8. Drag a Move block to the upper Sequence Beam of the Switch block, and then set the Direction item to **Stop**.
9. Add a Sound block after the Move block. Select **Goodbye** from the list of files.
10. Add a Stop block after the Sound block. (The Stop block is in the Flow group on the Complete Palette.)

Figure 6-21, Figure 6-22, and Figure 6-23 show the additions to the Loop block and the Configuration Panels for the Move and Sound blocks. (There are no configuration items for the Stop block, so I won't show its Configuration Panel.)

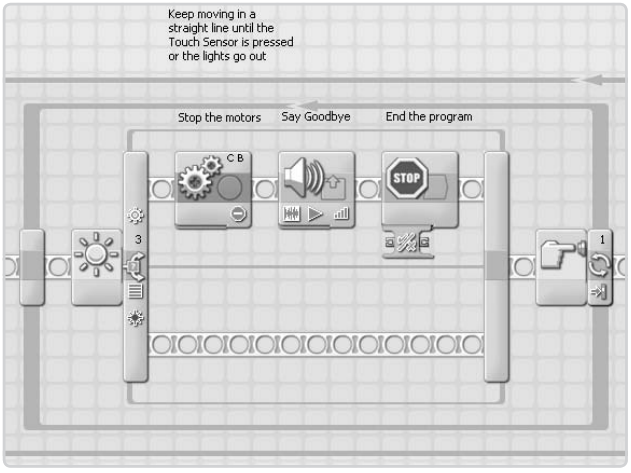


Figure 6-21: Looping until the Touch Sensor is pressed or the lights are turned out

Figure 6-22: Stopping the motors

Figure 6-23: Saying "Goodbye"

conclusion

The flow of your program is as important as the configuration of the blocks. You use the Switch block to make decisions, allowing you to choose between two groups of blocks. By nesting Switch blocks, you can extend the number of choices, as in the LineFollower2 program.

The other major flow control block is the Loop block, which allows you to repeat a group of blocks until a certain condition is reached. Much of the power of the NXT-G language comes from the flexibility you have in specifying the conditions for both of these blocks. The condition is usually based on a sensor reaching a set trigger value, although you can also configure the Loop block to repeat for a certain number of repetitions or a set amount of time.

You can also use the Stop and Keep Alive blocks to control the program flow. The Stop block allows the program to decide when it should end, and the Keep Alive block lets you prevent the NXT from turning itself off.

7

the WallFollower program: navigating a maze

In this chapter, I'll take you through the process of developing a fairly complex program using the parts of the NXT-G language that I've already covered. The WallFollower program that you'll create will allow the TriBot to navigate a simple maze. As you create the WallFollower program, I'll take you through the typical steps required to write a program, from the initial design to the final testing.

pseudocode

By the end of Chapter 6, the BumperBot program was a little long and complicated, and the WallFollower program will be just as complex. As you work with more complex programs, it becomes difficult to discuss them using English. English and other human languages are great for many types of communication, but they're not ideal for describing a program. It's difficult to give a short, precise description of a program in English.

When you write a program, you are creating the program's *source code*, or just *code*. Using NXT-G, this source code includes the arrangement of the blocks and their configuration.

You can describe how the program works using *pseudocode*. Pseudocode looks like a program but doesn't need to follow any strict rules. Pseudocode describes the most important details of how a program works and allows you to describe the logic behind the program and then translate it into whatever programming language you choose, including NXT-G.

For example, Figure 7-1 shows the RedOrBlue program from Chapter 5. The program waits for the Touch Sensor to be pressed and then uses the Light Sensor to tell whether the object being tested is blue or red. Listing 7-1 shows the pseudocode for this program.

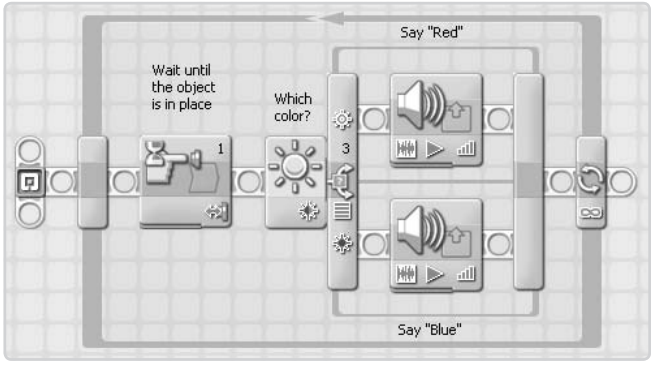


Figure 7-1: The RedOrBlue program

```
begin loop
   wait for the Touch Sensor to be pressed
   if Light Sensor > 42 then
      use a Sound block to say Red
   else
      use a Sound block to say Blue
   end if
loop forever
```

Listing 7-1: Pseudocode for the RedOrBlue program

As you can see, the pseudocode gives a concise, easy-to-understand description of the program. With a little practice, you'll quickly get used to reading pseudocode and turning it into a working NXT-G program.

There are a few things about Listing 7-1 to note:

* In most cases, a separate line is used for each block.
* The lines are indented to show how the blocks are nested (placed inside another block). Indenting makes it easier to see what's happening inside a Loop or Switch block, especially if they are nested.
* The Switch block is represented in the listing with a group of lines that include `if then else end if`, instead of something with the word *Switch* in it. This is a more common choice of wording because many programming languages use some type of `if-then` statement for a conditional. This is also closer to how you would describe the logic in English.
* The lines between `if Light Sensor > 42 then` and `else` describe the blocks on the Switch block's upper Sequence Beam, and the lines between `else` and `end if` describe the blocks on the lower Sequence Beam.
* The `end if` line marks where the Switch block ends, which can be helpful if you're writing by hand or on a white board and the lines are not indented perfectly.
* The trigger values and other settings are included in the listing. (You won't always have these values when you are first planning a program; just include whatever details you have.)
* In this example, the pseudocode shown reflects the final program and therefore is complete with many details filled in. Pseudocode is often used when first planning a program and during its initial development, and in these cases the code is usually less defined.

Listing 7-2 shows the pseudocode for the BumperBot3 program shown in Figure 7-2 and Figure 7-3. This program is much more complicated than RedOrBlue, but the pseudocode is still fairly easy to understand and offers a much more precise description of the program than one written in English. You should be able to read through this to follow the workings of this program.

```
begin loop
   Keep Alive block to prevent the NXT from going
      to sleep
   move forward, with duration unlimited
   begin loop
      if Light Sensor reads < 10 then
         stop the motors
         say Goodbye
         use Stop block to end the program
      end if
   loop until Touch Sensor is pressed
   stop the motors
   reset the Rotation Sensor for Motor B
   backup a slowly
   begin loop
      use a Sound block to beep for 0.5 seconds
      use a Wait Time block to pause for
         0.25 seconds
   loop until the Rotation Sensor for Motor B >
      300 degrees
   spin for 250 degrees
loop forever
```

Listing 7-2: Pseudocode for the BumperBot3 program

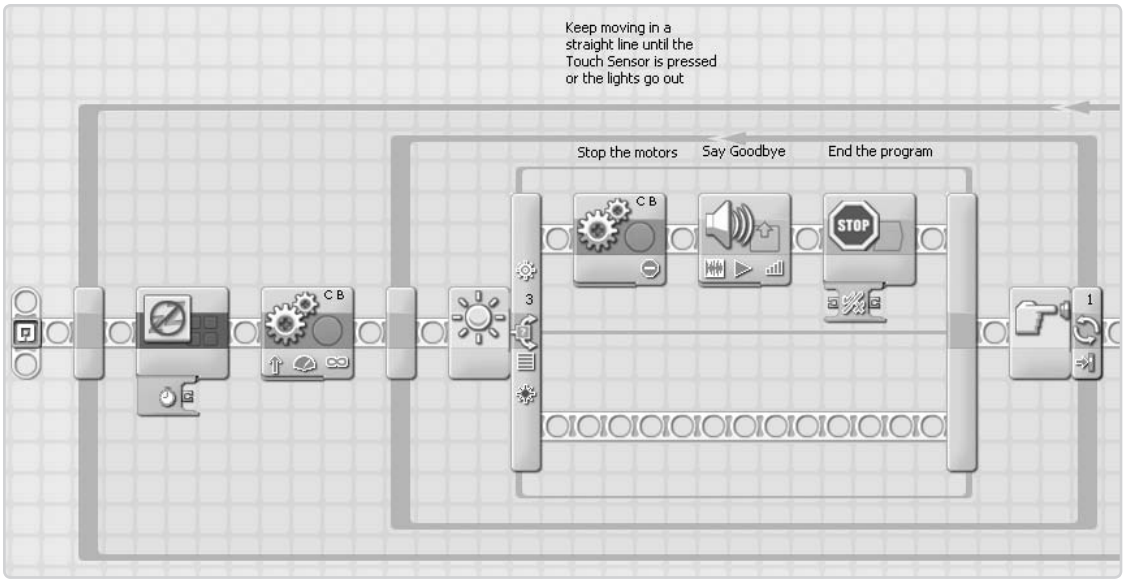


Figure 7-2: The BumperBot3 program, part 1

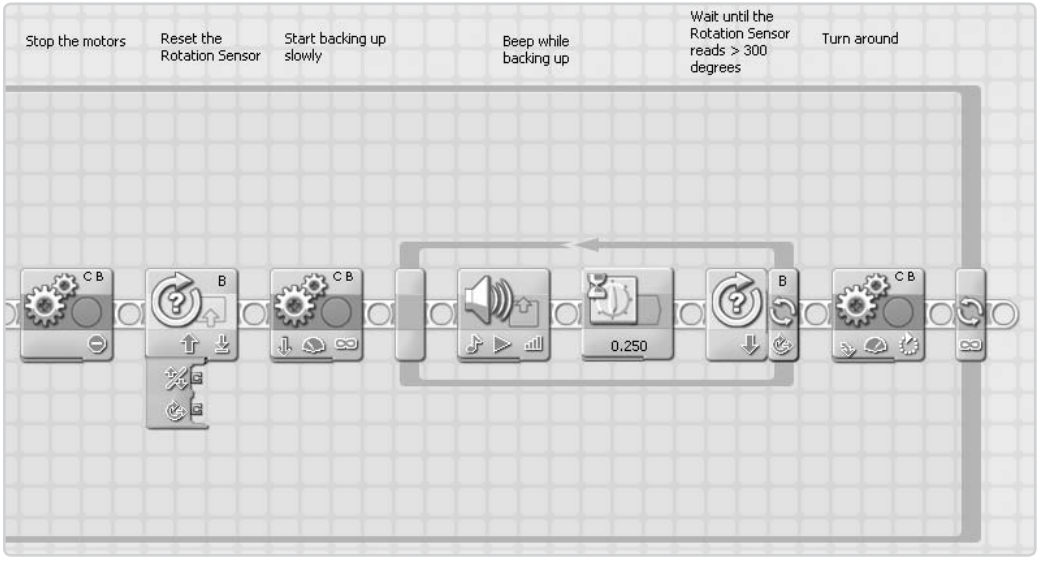


Figure 7-3: The BumperBot3 program, part 2

solving a maze

There are many well-known approaches to solving a maze. For this program, you will use a method known as the *right-hand rule* algorithm. (An *algorithm* is a set of instructions for solving a problem.) As the TriBot moves through the maze, it will always follow the wall to its right, going through any opening on that side.

The right-hand rule algorithm works for mazes without tunnels or bridges and where the start and end points are at the edges of the maze (this method won't work for a maze where the goal is to get to the center). As long as the maze fits these criteria, the robot is guaranteed to find the exit.

Figure 7-4 shows an example maze with the path followed using the right-hand rule. To help you understand how this rule works, imagine that you're walking through this maze, keeping your right hand on the wall. You would follow the path marked in the figure and eventually make your way to the exit. Although this method won't necessarily find the shortest route through the maze, it will eventually find the exit.

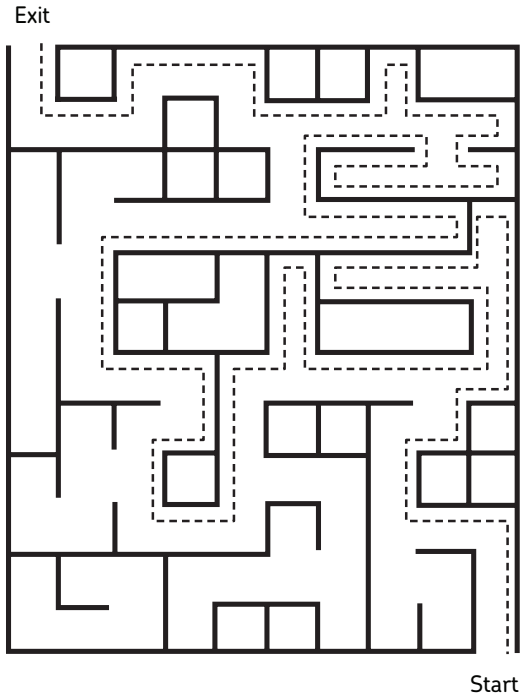


Figure 7-4: A simple maze and the right-hand rule path to the exit

program requirements

The first step in writing this program is to define what the program will do. To implement the right-hand rule algorithm, the program needs to follow a wall to the right of the TriBot, moving into any openings on that side. The TriBot also needs to be able to detect when it has reached a corner, at which point it needs to make the appropriate turn.

It's useful to create a simple list, called the *program requirements*, to describe exactly what the program needs to do. As long as the final program meets all the requirements on the list, then it should solve the original problem. Once you have the program written, you can use this list to test your design.

To develop the list of requirements, it's useful to think about the different situations the robot will encounter and decide how the program should react. For example, when there is a wall to the robot's right, as shown in Figure 7-5, it should keep moving straight.

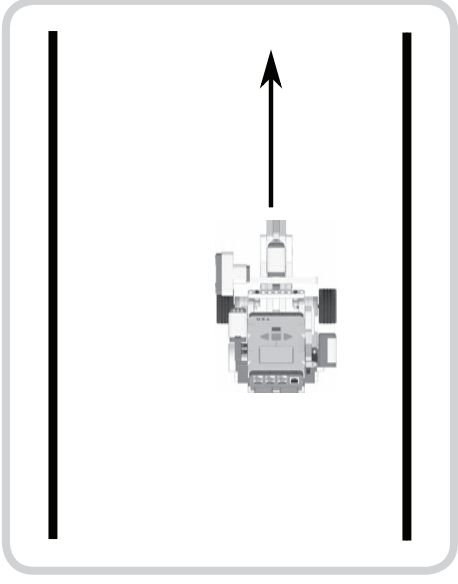


Figure 7-5: Following the wall to the right

When the wall to the right meets the wall in front at a corner, as shown in Figure 7-6, the robot should turn to the left and then continue forward.

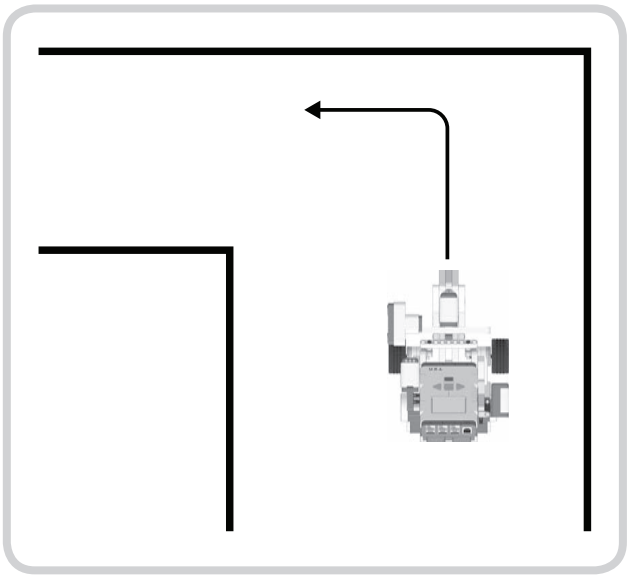


Figure 7-6: Turning left at a corner

When there is an opening in the wall to the right, as shown in Figure 7-7, the robot should turn into the opening. To follow the wall on its right, the robot should *always* turn into an opening, even if it could go straight as in Figure 7-8 or turn to the left as in Figure 7-9.

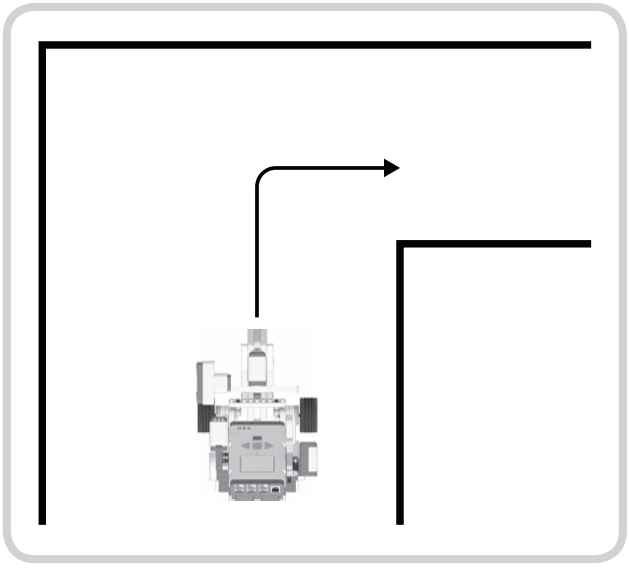


Figure 7-7: Turning into an opening on the right

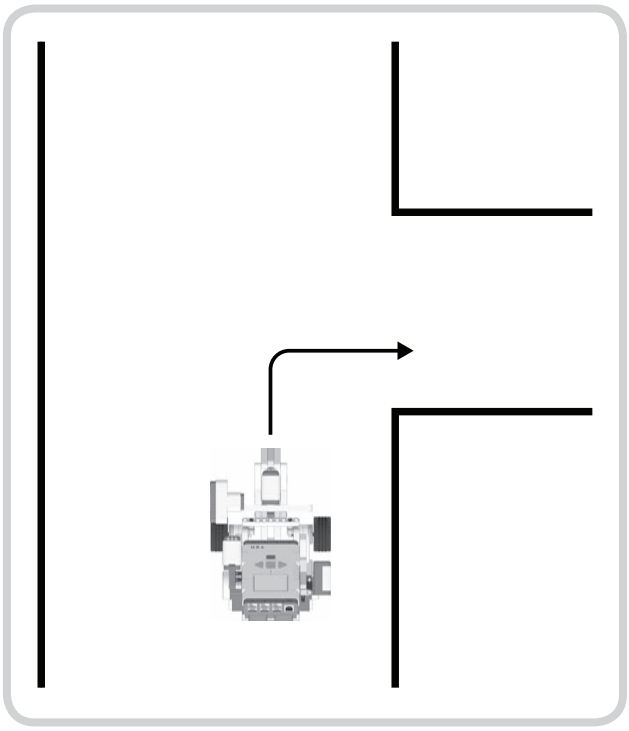


Figure 7-8: Turning right instead of going straight

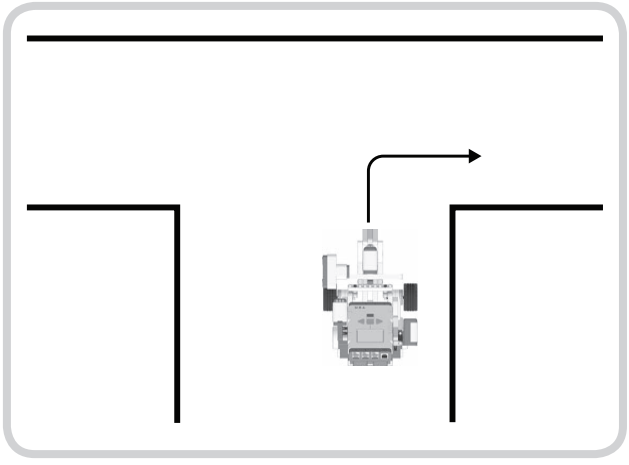


Figure 7-9: Turning right instead of turning left

Based on the previous analysis, you can summarize the requirements for the program with the following three statements:

* The TriBot will move forward along the wall on its right, staying close to the wall.

* If the TriBot finds a wall in front and to the right, it should turn left 90 degrees and then follow the new wall.
* If the TriBot comes to an opening in the wall on the right, it should turn right 90 degrees and go through the opening.

assumptions

While coming up with the requirements, it's also a good idea to list any assumptions or restrictions about the program. This helps you know which conditions you need to test and which you can ignore. You can make four assumptions for this program:

* The walls are straight.
* Any opening is big enough for the TriBot to fit through.
* Walls meet at right angles. This will make the turning at a corner easier.
* When the program begins, the TriBot will be placed against the wall.

The last item in this list, dealing with how the robot is arranged when the program starts, is called an *initial condition*. Starting the program with the robot in place will be much easier than making the robot wander around looking for the maze entrance.

Thinking about the assumptions before you start programming helps you to know which problems you need to solve and which you can ignore. When designing a program, it's helpful come up with the list of requirements and assumptions at the same time. The two lists together define what the program will do and what you don't expect it to do.

If your final program extends beyond your initial ideas, that's fine. For example, one of the assumptions is that the walls are straight. You may end up with a program that happens to work great with curved walls. The assumptions aren't things the program *can't* do; they are things it's not *required* to do.

initial design

The next step in designing a program is to think about the major tasks the robot has to perform and decide how you can solve them. For one thing, the TriBot should travel next to the wall while staying a short distance away from it. You can use the Ultrasonic Sensor to tell how far the TriBot is from the wall and use Move blocks with different steering settings to move toward or away from the wall.

Because the wall will be next to the TriBot as it's moving, you need to mount the Ultrasonic Sensor pointing to the side (instead of to the front). Follow the building instructions in "Alternate Placement for the Ultrasonic Sensor" on page 43 to mount the sensor so that it points to the side, as shown in Figure 7-10.

Figure 7-10: The TriBot with the Ultrasonic Sensor pointing to the side

The program needs to be able to tell when the TriBot has run into a corner. The Touch Sensor will handle this job nicely. When the TriBot enters a corner, it will run into the wall, and the Touch Sensor will be pressed, at which point the TriBot can back up, make a quarter turn to the left, and follow the wall it just ran into. (The BumperBot program has code similar to this, so you should have a good idea of how this will work.)

The program also needs to deal with openings in the wall. For this you'll use the Ultrasonic Sensor to tell when the robot passes an opening. If the sensor suddenly reads a large distance while the TriBot is following a wall, then you know that it has reached an opening.

Finally, the TriBot should keep moving until you stop it, so this will be another program that is in a loop.

Now that the high-level tasks are defined, you can write some pseudocode to describe the program (see Listing 7-3). Because you are at the early stage of developing the

program, the pseudocode will just cover the main points. In the following sections, you'll take each of the main parts, one at a time, and develop the NXT-G code.

```
begin loop
  if too close to the wall (use Ultrasonic
     Sensor) then
     steer away from the wall
  else
     steer toward the wall
  end if
  if Touch Sensor is pressed then
     back up a little to get room to turn around
     spin one quarter turn
  end if
  if an opening is detected (use Ultrasonic
     Sensor) then
     spin one quarter turn toward the opening
  end if
loop forever
```

Listing 7-3: Initial design for the WallFollower program

following a straight wall

The first section of NXT-G code will make the TriBot travel along the wall. You'll use a Switch block to choose between two Move blocks, one that turns toward the wall and one that turns away from the wall. As the TriBot moves forward, this should keep it about the same distance from the wall, as in the first version of the LineFollower program.

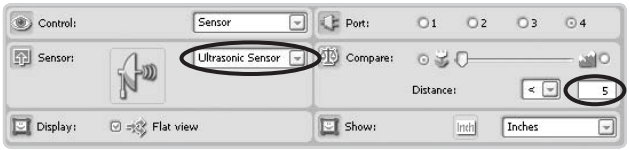
writing the code

Of course, you need to decide what the TriBot's distance from the wall should be. The TriBot should stay close enough to the wall while still having enough room to turn when it gets to a corner. You can get a reasonable starting value by using the NXT's View menu and following these steps:

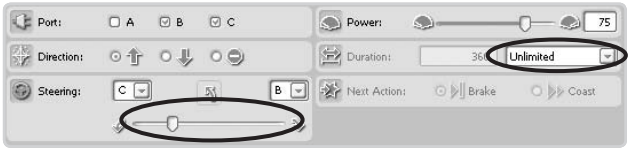
1. Put the TriBot on the floor next to the wall, and make sure there is enough room to spin the TriBot all the way around.
2. Adjust the robot so that the Ultrasonic Sensor is facing the wall.
3. Using the NXT's **View** menu, select either **Ultrasonic Inch** or **Ultrasonic cm**.
4. Set the Port item to **4.**
5. Note the value on the NXT's display. I got a reading of 5 inches (13 cm), so that's what I'll use in the following instructions.

Now you can start writing the program. Begin by putting the blocks together with some reasonable values and then fine-tune the settings after some testing:

1. Create a new program named *WallFollower*.
2. Drag a Loop block onto the Sequence Beam. The default Control value is **Forever**, which is what you want, so don't make any changes to this block.
3. Drag a Switch block into the Loop block.
4. Select the Ultrasonic Sensor from the Sensor list.
5. Set the Distance setting to the target value. The Configuration Panel for the Switch block should look like this:

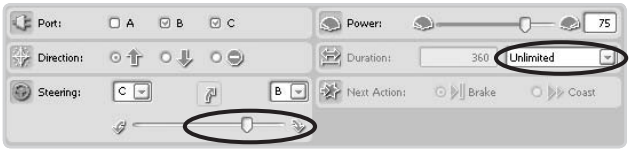


6. Drag a Move block onto the Switch block's upper Sequence Beam.
7. Set the Duration item to **Unlimited** to keep the motors moving while the steering is adjusted each time through the loop.
8. This block will be run when the Ultrasonic Sensor reads less than the trigger value, meaning the TriBot is too close to the wall. In this case, the robot should steer away from the wall. The Motor on the side of the TriBot farthest from the wall is plugged into port **C**, so set the Steering slider toward that motor. The Configuration Panel should look like this:



9. Drag a Move block onto the lower Sequence Beam of the Switch block.
10. Set the Duration item to **Unlimited**.

11. This block will be run when the TriBot needs to move toward the wall, so set the Steering slider toward the **B** motor. The two Move blocks behave the same, just steering in opposite directions. As you adjust these blocks during testing, remember to change both of them. The Configuration Panel for this block should look like this:



At this point, the program (shown in Figure 7-11) should let the TriBot travel along a wall. The next step is to do a little testing and modify the program as needed to get it to work correctly.

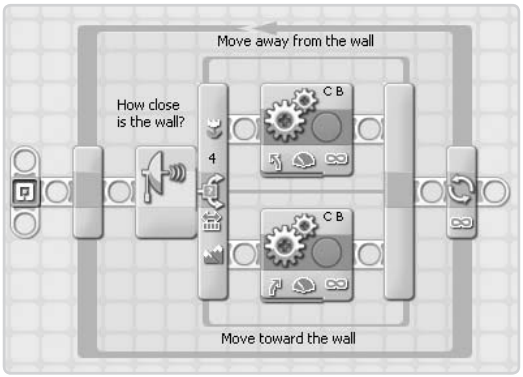


Figure 7-11: Following a wall

testing

The code presented previously will likely have a few problems. In this section, you'll do some testing and make some adjustments to fix the problems. You'll go through this process with every program that you write; it's almost impossible to get all the settings right without some testing.

To test the WallFollower program, you'll need a wall with a corner and an opening, or you can use a full maze, which is a little more fun. To build a maze, you just need walls that are tall enough so that the Ultrasonic Sensor detects them but doesn't see any objects beyond them. (It's not fair if the robot can "see" over the walls.) Spare lumber, boxes, or piles of books work well for the walls of the maze. You could even build a maze out of LEGO blocks, but it would take a lot of blocks (and a lot of time).

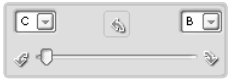
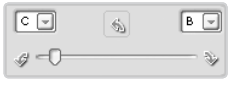
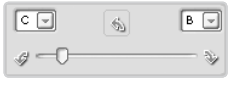
When you run the WallFollower program, the robot will probably quickly run into the wall or wander off into the middle of the room. One problem may be that the steering control is incorrect or that the steering is not being adjusted quickly enough. With the Power setting of the Move blocks set at the default value of 75, the TriBot moves pretty quickly. Slow the robot down to make adjusting the steering a little easier.

12. Set the Power setting to **35**. Remember to change both Move blocks.

Slowing the TriBot down in this way should help a lot; the TriBot should now make it farther down the wall, though it may eventually turn toward the wall and crash into it before turning away.

To avoid hitting the wall, set the Steering control to turn more quickly by moving the slider close to the end. The Steering slider has 10 steps between the middle and the end of the slider. Begin at the end of the slider and move toward the middle, testing each position to see which works best. Table 7-1 shows my results.

table 7-1: steering test results

steps from the end of the slider	steering control	results
1		The TriBot stays close to the wall without hitting it. The motion is very choppy, with a lot of side-to-side motion.
2		The TriBot stays close to the wall without hitting it. The motion is noticeably smoother than the previous setting. It's still not perfectly smooth, but it's not bad.
3		The TriBot does fine for a while but eventually runs into the wall.

Based on my results, I suggest setting the Steering slider to two steps from the end, though your results may vary. Figure 7-12 and Figure 7-13 show the Configuration Panels for the Move blocks with the changes. At this point, the TriBot should do a good job following a straight wall with no corners or openings.

Figure 7-12: Moving away from the wall

Figure 7-13: Moving toward the wall

turning a corner

The next part of the program uses the Touch Sensor to detect when the robot reaches a corner, at which point it turns the TriBot so that it can follow the new wall. This is similar to the BumperBot program, which has the TriBot back up and turn when it runs into something.

Listing 7-4 shows the pseudocode for this section of the program.

```
if the Touch Sensor is pressed then
  stop the motors
  backup far enough to turn the robot
  spin a quarter turn
end if
```

Listing 7-4: Turning a corner

After the TriBot backs up and turns, it needs to be positioned the correct distance from the wall, or it might run into the wall or wander away from it. To make this section of code work, you have to determine the correct Duration settings for the two Move blocks. Recall that in Chapter 4 you did some testing to determine the settings for the AroundTheBlock program; you can use those results as starting values and then adjust the values as needed after some testing.

writing the code

Follow these steps to add this section to the program:

13. Drag a Switch block into the Loop block, placing it to the right of the existing Switch block. The program should look like Figure 7-14. Keep the default settings for this block since they use the Touch Sensor. The blocks on the upper Sequence Beam of the Switch block will be used if the robot runs into a wall and the Touch Sensor is pressed.
14. Drag a Move block to the upper Sequence Beam of the new Switch block. Set the Direction item to **Stop**.
15. Add another Move block to the upper Sequence Beam; this block will make the TriBot back away from the corner.
16. Select the downward-pointing arrow for the Direction item.
17. Set the Duration item to **150 degrees** (remember to select **Degrees** first and then enter **150**).
18. Set the Power item to **35** to match the other Move blocks. The robot looks smoother when all the Move blocks use the same Power setting. (If you prefer jumpier motion, increase the speed of this block, and it should still be precise enough.)
19. Add another Move block to the upper Sequence Beam. This block will spin the TriBot so that the Ultrasonic Sensor is facing the new wall.
20. Set the Duration item to **235 degrees** and the Power item to **35**.

NOTE **The balloon tires are bigger than the flat tires, so you'll need to change some of the settings if you're using them. Use 180 degrees instead of 235 degrees for the Duration item.**

21. The robot needs to spin toward the C motor, so move the Steering slider all the way to the left end.

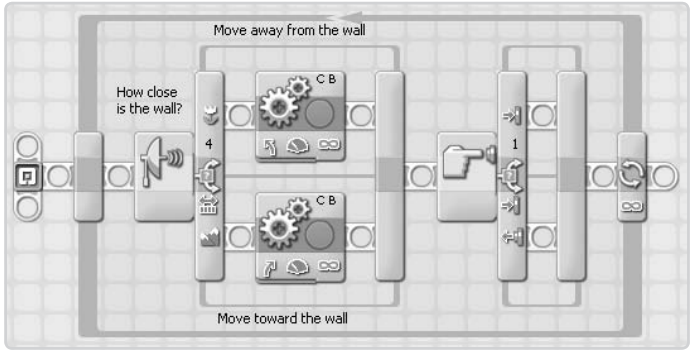


Figure 7-14: Adding another Switch block

22. This Switch block is using only the upper Sequence Beam, since the program doesn't need to do anything here if the Touch Sensor is not pressed. Deselect the **Flat view** option on the Switch block to make the program look a little cleaner.

Figures 7-15 through 7-19 show the new section of the program and the Configuration Panels for the Switch block and the three Move blocks.

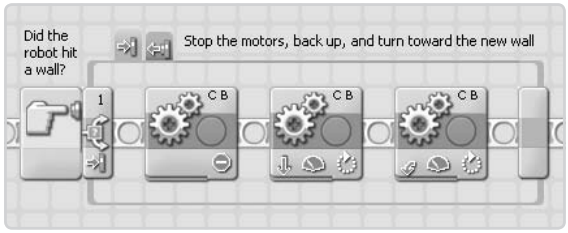


Figure 7-15: Turning a corner

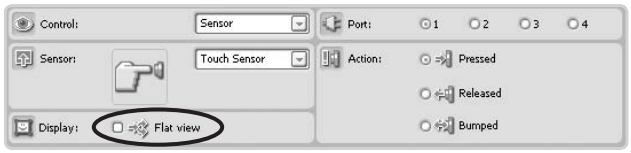


Figure 7-16: Did the robot hit a corner?

Figure 7-17: Stopping the motors

Figure 7-18: Backing away from the corner

Figure 7-19: Turning toward the new wall

testing

To test the new code, start the TriBot close to a corner and see how it reacts when it bumps into the wall. Figure 7-20 shows roughly how the robot should move as it backs up and turns around.

My initial testing revealed a couple of flaws: The TriBot doesn't move back far enough, and it spins a little too far. After trying a few values, I settled on backing up for 190 degrees and then spinning for 230 degrees. Figure 7-21 and Figure 7-22 show the Configuration Panels for the two Move blocks with the new settings.

NOTE **For the balloon tires, use 200 degrees instead of 230.**

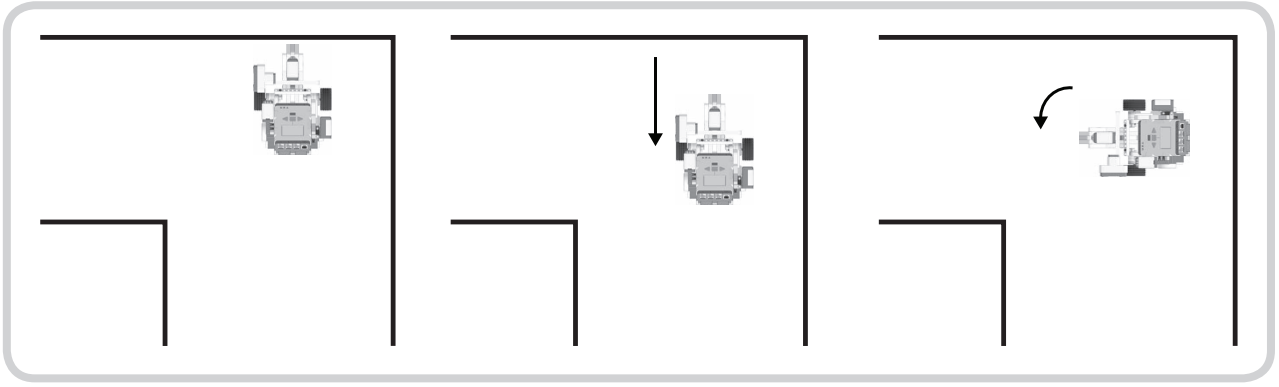


Figure 7-20: Backing away from the wall and turning to the left

Figure 7-21: New Duration setting for backing away from the corner

Figure 7-22: New Duration setting for turning

Before moving on to the next section of the program, it's important to retest the wall following coding to make sure you didn't accidentally break something while adding the new code.

NOTE **Whenever you add new code, it's a good idea to test the parts of the program that worked before you made changes. Most of the time everything will work fine, but if you did introduce a bug, then finding it sooner will make it easier to fix.**

going through an opening

When the TriBot comes to an opening in the wall, it should turn and go through the opening. You'll need to modify the code to recognize an opening in the wall and then turn the robot to move into the opening. Once through the opening, the TriBot should continue to follow the wall on its right.

The code you wrote earlier works well for following a straight wall, and it's helpful to see how it behaves when the TriBot reaches an opening. The existing program may fail, crashing the robot into the wall opposite the opening or sending the robot wandering in circles. But it's also possible that the program will work, following the wall around the corner of the opening and continuing to explore the maze. Seeing how the program responds to an opening will give you a better idea of exactly what changes need to be made for the TriBot to successfully go through the opening.

When I ran the TriBot at this point, I found that it actually did a pretty good job of turning and making it through an opening, but I wanted to improve things by making the turn into the opening a little tighter in order to keep the TriBot from straying too far from the wall.

When the TriBot gets to an opening, the Ultrasonic Sensor will suddenly read a much greater distance than it does when following the wall because the wall has fallen away. When the Ultrasonic Sensor first reaches the opening, the rest of the robot is still next to the wall, so it needs to move forward a little more before turning. Once the TriBot has spun a quarter turn toward the opening, the robot then needs to move forward a little more before the Ultrasonic Sensor will be next to the wall. Figure 7-23 shows how the TriBot should move, and Listing 7-5 shows the pseudocode for this section of code.

```
if the Ultrasonic Sensor detects an opening then
  stop the Motors
  move forward a little
  spin a quarter turn toward the opening
  move forward into the opening
end if
```

Listing 7-5: Moving through an opening

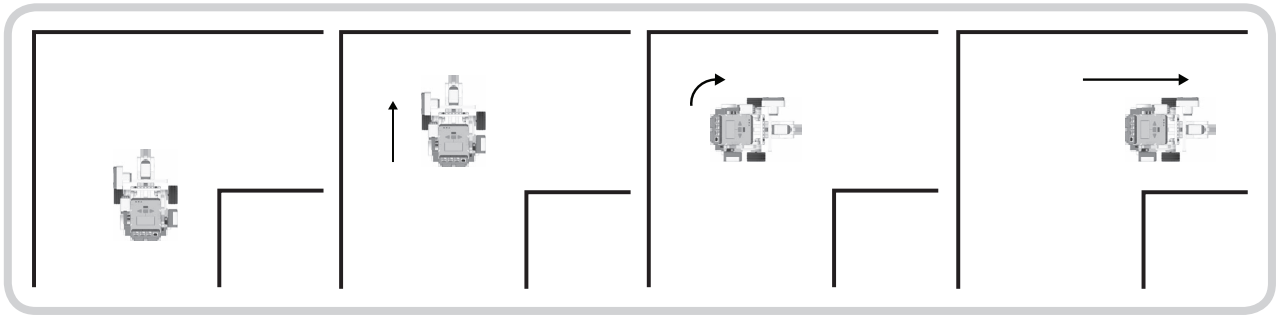


Figure 7-23: Turning and moving through the opening

Before adding any blocks to the program, you need to determine the values to use for the Switch block and the Move blocks. For the Switch block, begin with a trigger value of 10 inches (24 cm), which should be big enough to make sure the TriBot has found a real opening and not just a small bump in the wall. For the Move blocks, begin with the same value you used to back the TriBot away from the corner, and then adjust the value after some testing.

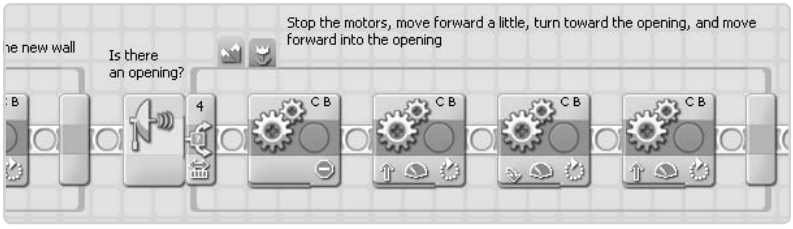


Figure 7-24: Turning into an opening

writing the code

Follow these steps to make the TriBot move through an opening.

23. Add a Switch block to the end of the code inside the Loop block.
24. Select the Ultrasonic Sensor. Set the Distance item to **10** and the comparison to **>** (greater than).
25. Deselect the **Flat view** option because you need to add blocks only to the upper Sequence Beam.
26. Add a Move block to the upper Sequence Beam of the Switch block. Set the Direction item to **Stop**.
27. Add another Move block; this one will move the robot forward so that it's all the way in front of the opening.
28. Set the Power item to **35** and the Duration item to **190 degrees**.
29. Add a third Move block. This one will spin the robot so that it's facing the opening.
30. Set the Power item to **35** and the Duration item to **230 degrees**.

NOTE **For the balloon tires, use 180 degrees instead of 230 degrees.**

31. Move the Steering slider all the way to the right end (toward the B motor).
32. Add a fourth Move block. This one will move the TriBot into the opening.
33. Set the Power item to **35** and the Duration item to **230 degrees**.

Figure 7-24 shows this part of the program. Figures 7-25 through 7-29 show the Configuration Panels for the Switch block and the four Move blocks.

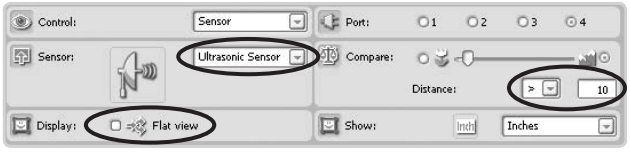


Figure 7-25: Detecting an opening using the Ultrasonic Sensor

Figure 7-26: Stopping the motors

Figure 7-27: Moving forward a little before turning

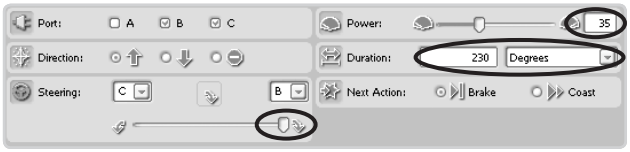


Figure 7-28: Turning to face the opening

Figure 7-29: Moving into the opening

using sound blocks for debugging

The section of code that notices an opening and then turns into it contains three Move blocks. When you're testing and changing this code, it can be useful to know when one move ends and the next begins so that if the robot's behavior isn't optimal, you'll have a better idea of which block to adjust.

One way to tell which block is running is to add Sound blocks before each block you are interested in. You can easily tell where each Move begins and ends by configuring each Sound block to play a different tone just before each Move block starts. (Be sure that the Sound block's Wait For Completion setting is selected so that the TriBot doesn't stop while it plays the sound.)

NOTE **Whenever you add code for debugging purposes, be sure that the added code has as little influence as possible on the program's timing; otherwise, when you remove the debugging code, the program may act differently.**

For this section of code, you may want to add a Sound block to the following places:

* Immediately after detecting the opening. If the opening is being detected too soon or too late, you can adjust the trigger for the Ultrasonic Sensor.
* After each of the three Move blocks. This will let you know where each move begins and ends so you know which block's Duration setting to adjust if the TriBot does not end up in the correct position.

Figure 7-30 and Figure 7-31 show how this section looks with the Sound blocks added, as well as the Configuration Panel for one of the blocks. The only difference between the four Sound blocks is the tone played.

When you are done testing, you can remove the Sound blocks, but be sure to test the program again after removing them to make sure that the program still works.

Figure 7-31: Sound block's Configuration Panel

testing

Test the new code by placing the TriBot along the wall near an opening and running the program. This is a complex program with several Move blocks, so many things can go wrong. Observe how the robot moves into the opening and starts following the new wall. Here are some things to look for:

* If the target value for the Ultrasonic Sensor is too large, the robot may not notice an opening.
* If the target value for the Ultrasonic Sensor is too small, the robot will turn toward the wall when there isn't an opening.
* If the robot does not move forward far enough, it will turn too soon and hit the corner of the wall.
* If the robot moves too far forward before turning, it will end up too far from the wall.
* If the turn is too short, the final move will make the TriBot move too far from the wall.
* If the turn is too long, the TriBot will run into the wall.
* If the final move is too short, the TriBot will end up outside the opening, and the Ultrasonic Sensor won't detect the wall.
* If the final move is too long, the TriBot can move past a tight corner or run into another wall.

My testing showed that the durations for all three moves were a little too short. After some experimentation, I found that setting the Duration item to 300 degrees for the move forward, 250 degrees for the turn, and 375 degrees

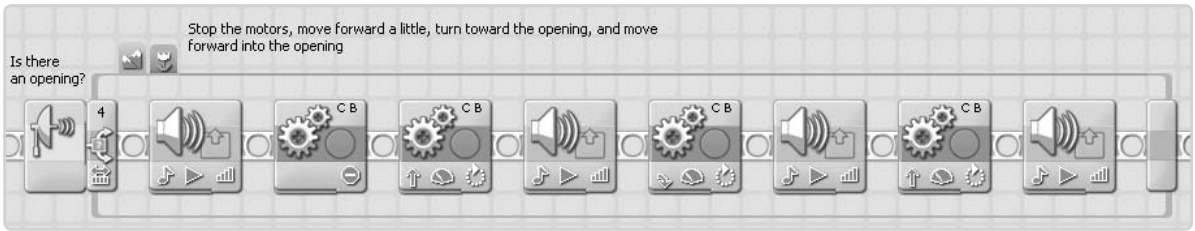


Figure 7-30: Adding Sound blocks for debugging

for the move into the opening works well. I also noticed that sometimes the TriBot acts as if there is an opening when there isn't one. Setting the target value for the Ultrasonic Sensor in the Switch block to 13 inches (33 cm) is a big improvement.

NOTE **If you're using the balloon tires, try setting the durations to 275, 210, and 325 degrees.**

Figures 7-32 through 7-35 show the updated Configuration Panels for the Switch block and the Move blocks.

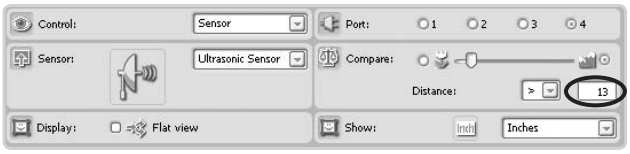


Figure 7-32: Updated target for the Ultrasonic Sensor

Figure 7-33: Updated duration for the move forward

Figure 7-34: Updated duration for the turn

Figure 7-35: Updated duration for the move into the opening

final test

Once you have the TriBot moving through an opening correctly, it's time to test the entire program. The requirements for this program (as listed in "Program Requirements" on page 86) state that the TriBot should be able to follow a straight wall and negotiate a corner or opening. If the TriBot can do these things, then it should be able to navigate a simple maze. Try a variety of test mazes to see whether the program correctly handles several turns and corners in a row.

In addition to making sure the program works as designed, it's interesting to see what else it can do and what will make it will fail. For example, try increasing the speed of the Move blocks to see how fast the robot can move before it starts to have problems, or adjust the spacing of the walls to see how that influences the robot's behavior.

It's also a good idea to check the assumptions made during the initial design. In fact, sometimes you'll find that your final result works better than you initially planned. For example, I found that I could relax the assumption about needing a straight wall; the program seems to handle slight curves and angles without a problem.

conclusion

In this chapter, I've taken you through all the steps involved in developing a typical NXT-G maze-navigating program. You've developed this program piece by piece, just as you would develop a program on your own. At each step you added a logical piece of the program, did some testing, and then made necessary modifications to adjust the move durations and to fix bugs.

The next chapter starts your exploration of data wires, one the most powerful features of the NXT-G language.

8

data wires

In this chapter, you'll learn how to use *data wires* to pass information from one block to another. Using data wires, you'll be able to change a block's settings while a program is running, and you'll be able to modify a sensor's value before it's used by another block. Data wires are one of the most powerful features of the NXT-G language, and learning how to use them will greatly enhance your skill as a programmer.

I'll begin with a simple example to show you what a data wire is and how it works. Most of the remainder of the chapter is devoted to developing a fairly complex program that turns the TriBot into a sound generator. Along the way, I'll cover all the basic concepts you need in order to successfully use data wires in your own programs. I'll also introduce several new blocks that are particularly useful when working with data wires.

what is a data wire?

Most blocks require information, or *data*, to perform some sort of action. For example, the Move block needs to know which motors to use, how fast to move them, and for how long, before it will actually turn the motors. This data is called the block's *input data*. You've used the Configuration Panel to set each block's input data in all the programs you've written so far in this book.

Some blocks create data for use by other blocks. For example, the Ultrasonic Sensor block reads data from the Ultrasonic Sensor and provides that information to other blocks. This data is called the block's *output data*.

A data wire takes output data from one block and uses it as the input data for another block. This gives you a lot more flexibility than just using the Configuration Panel, because you can change a block's settings while the program is running. For example, you can use the output from a sensor to control another block.

the GentleStop program

The GentleStop program is designed to show you how to use a data wire. The program moves the TriBot forward and then makes it gradually slow down and then stop when it reaches a wall.

Key to this program is that the speed of the TriBot depends on how far it is from the wall, as shown in Figure 8-1. As the robot moves closer to the wall, its speed smoothly decreases until it stops. To accomplish this feat, you'll combine the features of an Ultrasonic Sensor block and a Move block. The Ultrasonic Sensor block will measure how far the robot is from the wall, and the Move block will use this value for its Power setting to control the robot's speed. Figure 8-2 shows how these two blocks will be connected in the program.

As the program runs, the Move block's Power setting will be adjusted according to the robot's distance from the wall as measured by the Ultrasonic Sensor. For example, when the robot is 80 cm from a wall, the Power setting will be 80, and when it's 20 cm from the wall, the Power setting will be 20. When the TriBot gets very close to the wall, usually around 7 cm (3 inches), the Power setting will be too low to move the robot, and it will come to a stop.

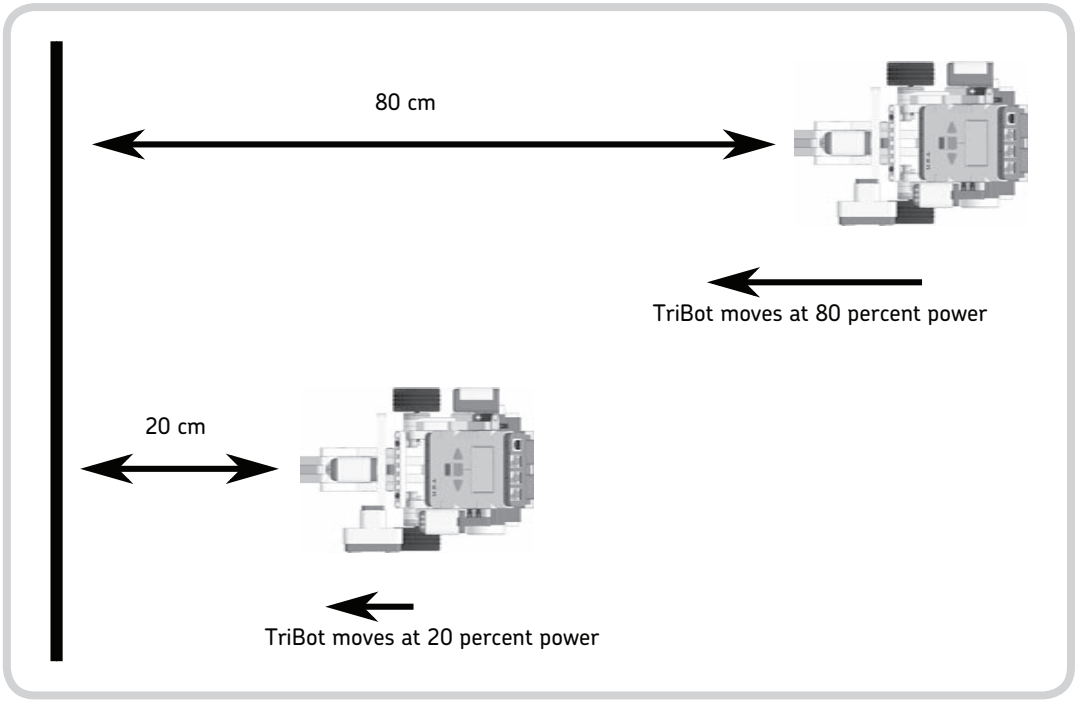


Figure 8-1: The TriBot moves slower as it approaches the wall.

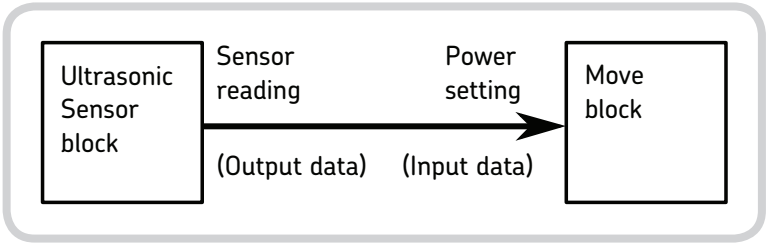


Figure 8-2: Combining the Ultrasonic Sensor and the Move block

Figure 8-3: The Ultrasonic Sensor pointing to the front of the TriBot

It's possible for the Ultrasonic Sensor block to generate a value that's larger than the maximum Power setting of 100, because the Ultrasonic Sensor can measure distances up to 255 cm. NXT-G will handle this situation gracefully, and the Move block will use 100 for the Power setting if the value from the Ultrasonic Sensor block is greater than 100.

NOTE To measure the distance to a wall, the Ultrasonic Sensor must be pointing forward. If yours isn't pointing forward, move it now, as shown in Figure 8-3.

You'll begin writing the program by putting an Ultrasonic Sensor block and a Move block inside a Loop block, and then you'll use a data wire to connect the output from an Ultrasonic Sensor block to the Power setting of a Move block.

Figure 8-4: The Ultrasonic Sensor block on the Complete Palette

Follow these steps to create the program:

1. Create a new program named *GentleStop*.
2. Drag a Loop block onto the Sequence Beam, and keep all the default settings.
3. Drag an Ultrasonic Sensor block from the Complete Palette (as shown in Figure 8-4) into the Loop block. (Display the Complete Palette by clicking the center tab at the bottom of the Programming Palettes.)

Figure 8-5 shows the Configuration Panel for the Ultrasonic Sensor block. The controls are the same as those used with the Wait, Loop, and Switch blocks. All the Sensor blocks (Touch Sensor block, Sound Sensor block, and so on) use the same controls you're familiar with from the program flow blocks. Use a Sensor block instead of a program flow block when you want to use a sensor to control something other than a Wait, Switch, or Loop block.

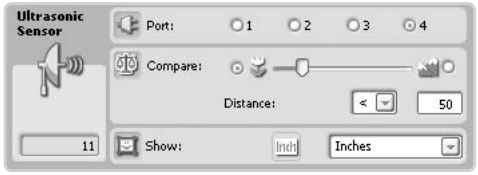
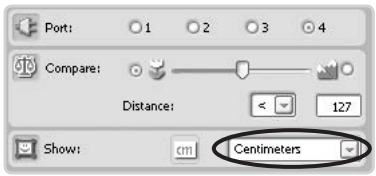


Figure 8-5: The Ultrasonic Sensor block's Configuration Panel

The Ultrasonic Sensor can report the distance using either inches or centimeters, depending on the Show setting. Using inches, the TriBot will begin slowing down when it is less than 100 inches from the wall, which is a fairly large distance. A centimeter is a smaller distance than an inch, so the TriBot will get closer to the wall before slowing down if you use centimeters (100 cm is about 39 inches). Letting the robot move at top speed until it's closer to the wall makes the program more interesting to watch.

4. Select **Centimeters** for the Show item.



5. Drag a Move block into the Loop block, and set the Duration item to **Unlimited**.

With the three blocks in place, the program should look like Figure 8-6. You are now ready to connect the data wire between the Ultrasonic Sensor block and the Move block. Data wires connect to an area of the block known as the *data hub*. The Move block's data hub isn't displayed when you add the block to the program, so before attaching a data wire, you need to open the data hub.

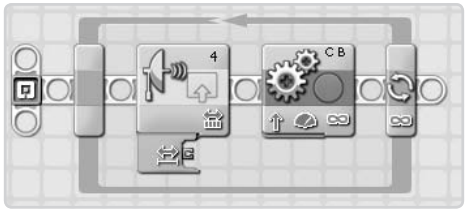
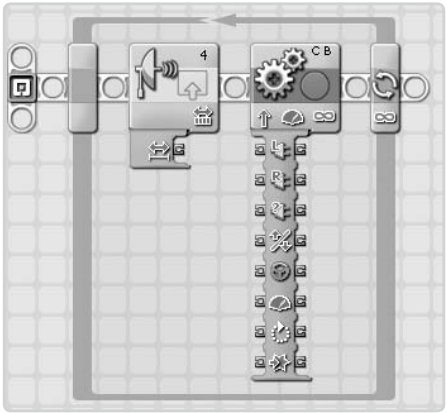


Figure 8-6: Before connecting the data wire

6. Open the Move block's data hub by clicking the tab at the bottom, as shown here:

7. The program should now look like this:



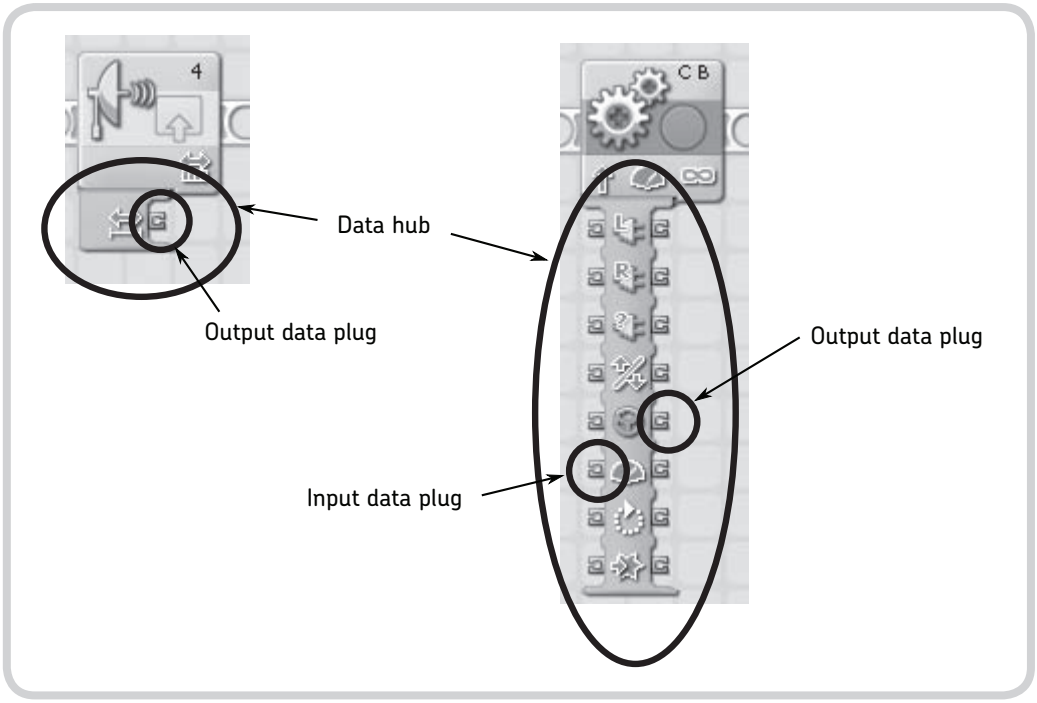
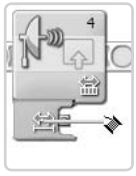


Figure 8-7: Data hubs and data plugs

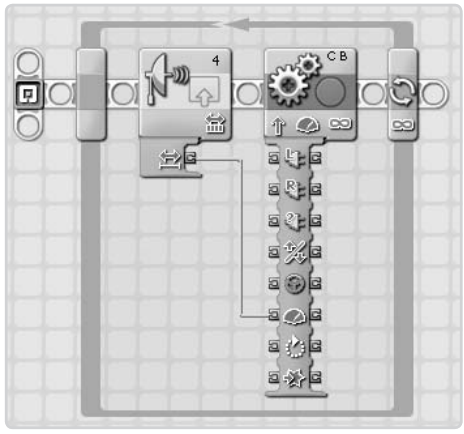
The data hub contains a number of *data plugs*. The data plugs on the left side of the data hub are used to pass data into the block, and those on the right side are used to pass data out of the block, as shown in Figure 8-7.

Now connect the Distance plug from the Ultrasonic Sensor block to the Move block's Power plug by following these steps:

8. Move the mouse cursor over the Ultrasonic Sensor block's output data plug. The mouse cursor should turn into a spool of wire, like this:
9. Click the mouse button. When you move the mouse, you should see a yellow wire between the mouse cursor and the data plug, like this:



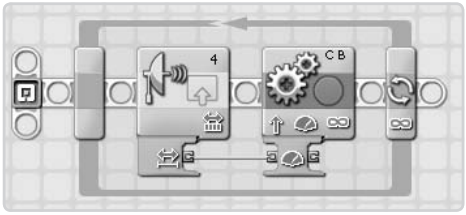
10. Drag the mouse (and wire) to the Move block's Power data plug, and click the mouse button to attach the wire. The Power data plug looks like this:
11. The data wire should now be connected to the two plugs, as shown here:



NOTE **If you accidentally connect the data wire to the wrong plug (which is easy to do), select Edit ▸ Undo (or press CTRL-Z) to erase the data wire and start again.**

12. To close the data hub and display only the connected plugs, click the tab at the bottom of the Move block, as shown here:

13. The program should now look like this:



Now it's time to download and test the program. To do so, start the TriBot in the middle of the room, and point it straight at a wall. The TriBot should start off quickly and then slow down as it gets closer to the wall. It should gently come to a stop just before hitting the wall, when the Power setting becomes too low to move the robot.

tips for drawing data wires

Usually connecting two blocks with a data wire is as simple as following the steps you used earlier for the GentleStop program. Occasionally you may want to delete a data wire or have more control over how the wire is drawn. Here are some tips to make life a little easier when working with data wires:

* Move slowly and wait for the mouse cursor to change to the wire spool before drawing the date wire.
* To erase a wire while drawing it, press the ESC key.
* Press the spacebar while drawing a data wire to change the wire's bend.
* You can erase a wire immediately after drawing it by selecting Edit ▸ Undo.
* To delete a data wire, click the data plug at the right end of the data wire. (In the GentleStop program, this would be the data plug of the Move block.)
* Deleting a block will also delete all data wires connected to it.

A data wire will automatically bend around any intervening blocks or data hubs as you draw it; this is a process known as *automatic routing*. Automatic routing usually results in a very clean-looking program with short data wires. However, sometimes a program with many data wires may be drawn with data wires overlapping or crisscrossing in ways that make it difficult to tell where individual wires begin and end. In this case, you may want to use *manual routing* to specify exactly how the wire should be drawn.

To use manual routing, click the mouse as you draw the data wire to fix its position and create a bend, effectively putting a virtual staple in that wire at a particular point. You can create as many bends in a data wire as necessary. Although it's not always possible to completely avoid crisscrossing wires, minimizing the number of crossing wires will make your programs much easier to understand.

NXT-G will redraw data wires as needed, as data hubs are opened or closed, or when blocks are added or deleted. As part of this redrawing process, any data wires you routed manually will be redrawn using automatic routing. Therefore, you may want to write and test your program first and then delete and manually route any potentially confusing data wires. It's a bit annoying to meticulously draw several data wires only to have them redrawn when you add a new block.

the SoundMachine program

The next program, SoundMachine, turns the TriBot into a simple sound generator. The wheel attached to motor B controls the volume; turn it to make the sound louder or softer. The wheel attached to motor C controls the tone (or pitch); turn it to make the sound higher or lower.

The program uses a Sound block to create the sound and uses two Rotation Sensor blocks to measure how far

each wheel has been turned. You'll use data wires to connect the output from the Rotation Sensor blocks to the Sound block so that the B motor controls the volume and the C motor controls the tone. I'll present the program in three parts; first you'll control only the volume, then you'll add some code to control the tone, and finally you'll display the volume and tone values on the NXT's screen.

controlling the volume

The first part of the program will look similar to the GentleStop program, only using Rotation Sensor and Sound blocks instead of Ultrasonic Sensor and Move blocks. You'll configure the Sound block to play a tone and use the value from the Rotation Sensor to control the volume.

Listing 8-1 shows the pseudocode for this part of the SoundMachine program. The entire program is contained in a loop, which, in turn, contains the Rotation Sensor and Sound block connected to each other using a data wire.

```
begin loop
   read the Rotation Sensor for motor B.
   use a Sound block to play a tone. Use the
      Rotation Sensor value for Volume.
loop forever
```

Listing 8-1: Controlling the volume

Figure 8-8 shows the program before connecting the data wire. The Rotation Sensor block is configured to use the B motor, and the Sound block has the Action set to Tone. The Sound block's Wait for Completion option is unselected so that the program won't pause while it plays the sound. The loop repeats so that the volume can be adjusted while the sound is playing. Figure 8-9 and Figure 8-10 show the Configuration Panels for the Rotation Sensor and Sound blocks.

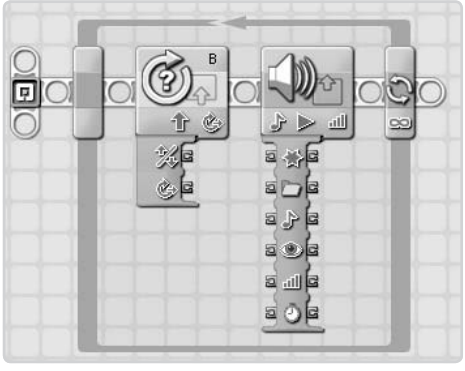


Figure 8-8: The SoundMachine program before connecting the data wire

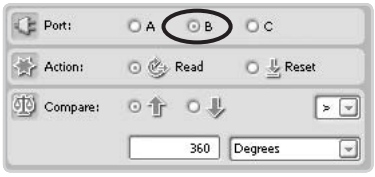


Figure 8-9: Configuration Panel for the Rotation Sensor

Figure 8-10: The Sound block's Configuration Panel

Connecting the Rotation Sensor's Degrees plug to the Sound block's Volume plug completes this part of the program, as shown in Figure 8-11. Once the data wire is connected, the Sound block will ignore the Volume setting in the Configuration Panel. Even though the Configuration Panel shows 75 for the Volume setting, when the program runs, the block will use the value from the data wire. When looking at a program to figure out why a block is behaving a certain way, it's important to consider both the data wires and the Configuration Panel settings, because the data wires will override the Configuration Panel settings.

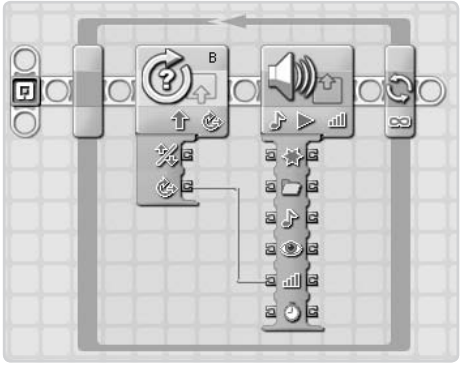


Figure 8-11: The wheel rotation controls the volume.

When the program starts, you won't hear anything because the Rotation Sensor will read 0, but turn the B motor, and the sound should get louder.

The Sound block's Volume setting uses values from 0 (no sound) to 100 (the loudest setting). The value from the Rotation Sensor block is measured in degrees, and 100 degrees is a little more than a quarter turn of the wheel. This means you can adjust the volume from the quietest to

the loudest setting by turning the wheel just past a quarter of a rotation. (The volume won't increase smoothly, but that's just how the Sound block works.)

In NXT-G 2.0, the Rotation Sensor block reports a negative value when a motor is rotated backward, whereas in NXT-G 1.1 the value is always positive or zero, and the Direction plug is used instead to tell the direction of a motor's rotation. When using NXT-G 2.0 with the SoundMachine program, you must rotate the wheel forward for the program to work. Rotating the wheel backward will cause the Rotation Sensor block to generate a negative number, and as a result, the Sound block won't make any noise because the Volume setting will be a negative number.

using the math block

For the next part of the SoundMachine program, you'll need to do a little math, which in NXT-G is accomplished using the Math block. You'll find the Math block on the Data group of the Complete Palette (Figure 8-12), and it should be displayed in your program as shown in Figure 8-13.

Figure 8-12: The Math block on the Complete Palette

Figure 8-13: The Math block

The Math block takes one or two numbers as input and an operation to perform on the numbers. You can enter the numbers into the Configuration Panel (Figure 8-14) or supply them via data wires. The result of the operation is available on an output data plug. All versions of the NXT-G Math block can perform addition, subtraction, multiplication, and division; the NXT-G 2.0 version can also calculate the absolute value and square root.

Figure 8-14: Configuration Panel for the Math block

adding tone control to the SoundMachine program

In this section, you'll add tone control to the SoundMachine program using the C motor as a dial. The Sound block's Configuration Panel (shown in Figure 8-15) makes selecting the Sound block's Note setting (or tone) very simple; you just click one of the keys on the small keyboard. Setting the tone using a data wire is a little different; for this, you need to use the Tone Frequency data plug (shown in Figure 8-16). This data plug takes a value measured in hertz (Hz), with a range from 264 (the lowest pitch) to 4000 (the highest pitch).

Figure 8-15: Setting the tone using the Configuration Panel

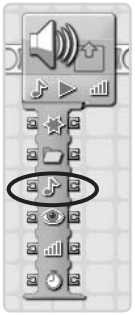


Figure 8-16: The Sound block's Tone Frequency data plug

NOTE Use the help file to learn about the values a data plug uses. The help file has a section for each block that includes a table describing all the data plugs the block supports. To open the help topic for a block, select the block, and press F1.

Controlling the Sound block's Tone setting is more complicated than controlling the Volume setting because the range of values used for the Tone Frequency setting is quite large. If you connect the Rotation Sensor block directly to the Tone Frequency data plug, like you did for the volume, then you'll need to turn the wheel about 11 full rotations to get to the highest pitch sound, which isn't very convenient.

You can solve this problem by using a Math block to multiply the value from the Rotation Sensor block by 40 before passing the value to the Tone Frequency data plug. This will make Rotation Sensor values between 0 and 100 (the same range you used for the volume) turn into Tone

Frequency values between 0 and 4000. (Rotation values less than 9 will produce Tone Frequency values that are too small for the NXT's sound system, but you can ignore this to keep things simple.)

The new code is similar to the code used for the volume control except that a Math block is needed between the Rotation Sensor block and the Sound block to multiply the value by 40. Here is the pseudocode for the program, with the new parts in bold:

```
begin loop
   read the Rotation Sensor for motor B
   read the Rotation Sensor for motor C
   use a Math block to multiply the motor C
      rotation by 40
   use a Sound block to play a tone. Use the Rotation
      Sensor value for Volume. Use the Math block
      result for the Tone Frequency
loop forever
```

Figure 8-17 shows the program with the changes. The new Rotation Sensor block and the Math block are placed between the existing Rotation Sensor block and the Sound block to avoid crossing the data wires. Figure 8-18 and Figure 8-19 shows the Configuration Panels for the Rotation Sensor and Math blocks.

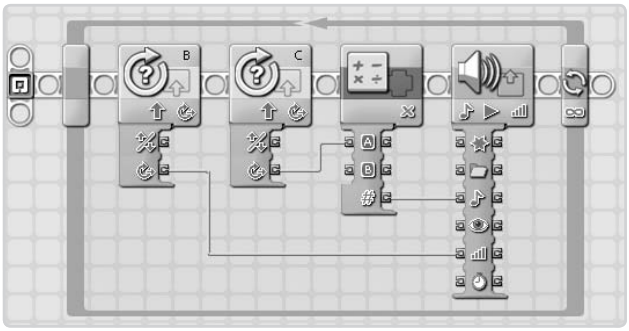


Figure 8-17: The SoundMachine program with tone control added

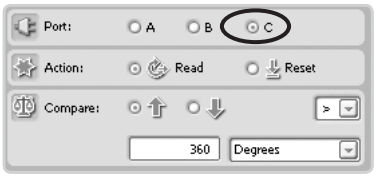


Figure 8-18: Configuration Panel for the second Rotation Sensor block

Figure 8-19: Multiply the input by 40.

When you now run the program, you should be able to control both the tone and the volume using the TriBot's wheels.

understanding data types

Before you finish the SoundMachine program, you need to learn about one more aspect of data wires. So far, all the data you have worked with (the Ultrasonic and Rotation Sensor readings and the Power, Volume, and Tone Frequency settings) has been numbers. But numbers aren't the only type of information. Think about the answers to the following three questions:

1. What is your name?
2. How old are you?
3. Are you the oldest child in your family?

Each question asks for a different kind of information. Your name is a word, your age is a number, and the answer to the third question is either "yes" or "no." In computer programming, the term *data types* describes different kinds of information. Your answers to the previous three questions correspond to the three data types that NXT-G supports:

- **Text values** are groups of characters that may include letters, numbers, and punctuation. For example, in the first program you created in this book, you used the Display block to print the text value `Hello`. In NXT-G programs, text values are used mainly for displaying information on the NXT's LCD display.
- **Numbers** are used to represent value readings from sensors and for setting trigger points. Numbers are also used for many other block settings, such as the Move block's Power and Steering settings.
- **Logic values** can be either true or false. For example, in addition to reading the distance value, the Ultrasonic Sensor block can compare the reading to a trigger value. The result of this comparison is a logic value; it's true if the

trigger value has been reached, and it's false if it hasn't. Depending on how a logic value is used, you may see it labeled as either True/False or Yes/No. This type of value is often called a *binary* value because it can have only one of two possible values.

Each data plug will work with only one data type. For example, you can only pass a number to the Move block's Power data plug (in other words, using the text value "really fast" won't work). The help file information about each block's data plugs includes the data type each plug expects.

Figure 8-20: The Number to Text block on the Complete Palette

Figure 8-21: The Number to Text block

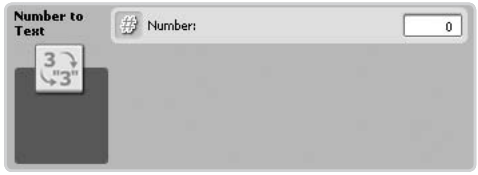


Figure 8-22: The Number to Text block's Configuration Panel

using the number to text block

For the next part of the SoundMachine program, you'll display the Tone Frequency and Volume values on the NXT's screen. Displaying the values your program uses can be a big help when debugging. There is one small complication: The Tone Frequency and Volume values are numbers, and the Display block can print only text values.

To print a number, you first need to convert the number to text using the Number to Text block (in the Advanced group at the bottom of the Complete Palette, as shown in Figure 8-20 and Figure 8-21). The Number to Text block takes a number as input and outputs the equivalent text data. The input value is usually supplied using a data wire, although it can be set in the Configuration Panel (Figure 8-22).

displaying the tone frequency

To display the Tone Frequency value, you'll use a Number to Text block to convert the value to text and then use a Display block to print the value. Figure 8-23 shows the SoundMachine program with the two new blocks added and the data wires connected.

Notice that the data wire supplying the input to the Number to Text block is connected to the Sound block instead of the Math block. The output side of the Tone Frequency data plug is called a *pass-through* data plug because it passes on the data supplied to the input side. Using the pass-through data plug from the Sound block is the same

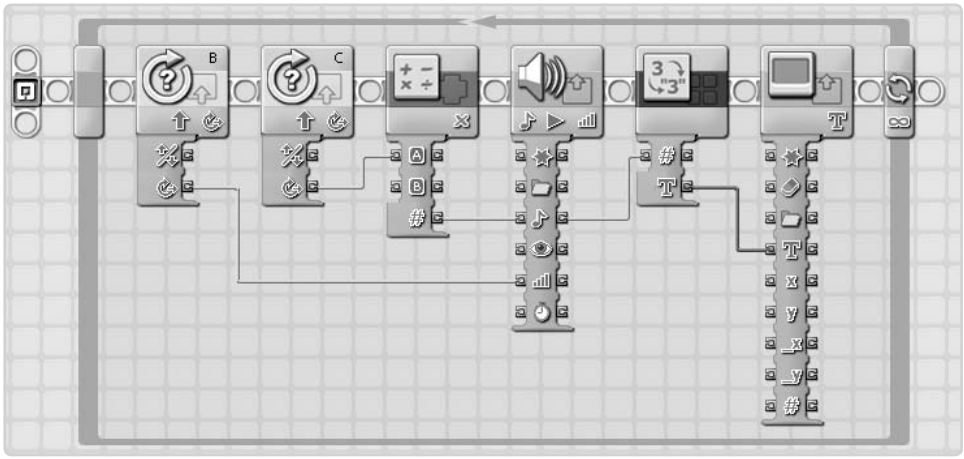


Figure 8-23: Displaying the Tone Frequency value

as adding a data wire from the Math block to the Number to Text block. The Sound block doesn't change the data while passing it on; the value passed into the left side of a plug is passed out on the right side unchanged. The pass-through plug can make a program cleaner looking and easier to understand.

NOTE **You can use a pass-through plug only when the input side of the plug is connected to a data wire. For example, you can't use the pass-through side of the Sound block's Duration plug because you haven't set the Duration value using a data wire. In other words, you can't use a pass-through if there is no input to "pass through!"**

Figure 8-24 shows the Display block's Configuration Panel block. Notice that you still need to set the Action setting to Text, even though you are supplying the text using a data wire. Once the data wire is connected, the Text value set in the Configuration Panel (*Mindstorms NXT*) will be ignored, and the value passed from the Number to Text block will be used instead.

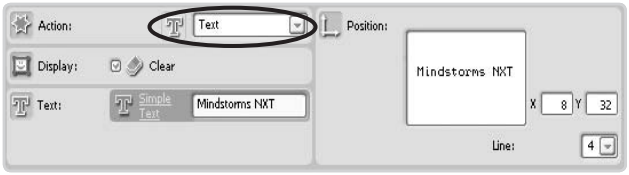


Figure 8-24: The Display block's Configuration Panel

When you run this version of the program, the value used for the tone frequency should be displayed on the NXT's screen.

RESTORING PASS-THROUGH DATA PLUGS

When the IDE redraws data wires, it will usually replace a data wire attached to a pass-through plug with one that starts at the original output plug. It actually does this in a couple of steps, and you can press CTRL-Z (undo) to "back up" one or more steps. For example, Figure 8-25 shows a section of the SoundMachine program where the output from the Math block passes through the Sound block to the Number to Text block.

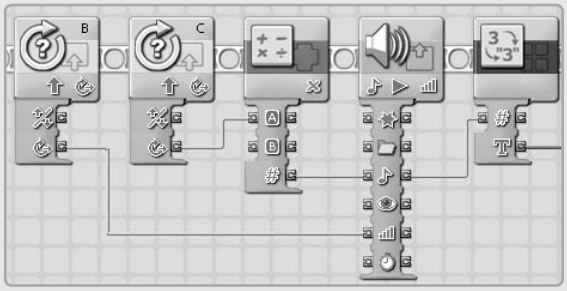


Figure 8-25: The original data wire routing

Figure 8-26 shows the blocks after closing the Sound block's data hub. The data wires have moved so that the one connected to the Number to Text block's input plug now connects directly to the Math block's output plug. The data wire connected to the Sound block's Volume plug has also been moved so that it now passes behind the Math block, making it appear as if the B input to the Math block is being used.

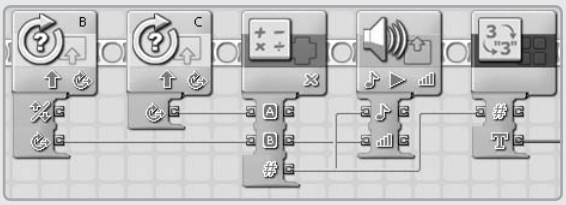


Figure 8-26: The data wires redrawn after closing the Sound block's hub

Figure 8-27 shows the blocks after pressing CTRL-Z, which puts the data wire connected to the Number to Text block's input plug back to the Sound block's output plug.

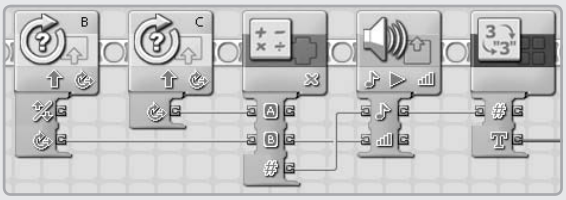


Figure 8-27: After pressing CTRL-Z

The arrangement of the data wires is still not ideal because the data wire for the Volume passes behind the Math block. You can remove this data wire redraw it with manual routing to avoid this potential source of confusion. Figure 8-28 shows the final configuration of the data wires.

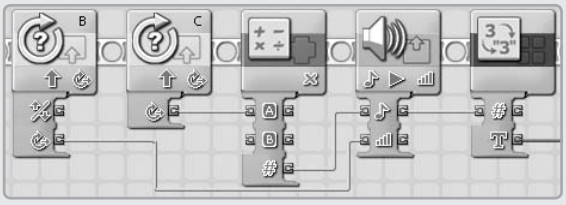


Figure 8-28: After fixing the data wire for the volume

using the text block

For the final set of changes to the SoundMachine program, I need to introduce the Text block. The Text block lets you join together up to three pieces of text, which can be useful for adding labels to values you display on the NXT's screen. You'll find the Text block in the Advanced group at the bottom of the Complete Palette, as shown in Figure 8-29. In your program, the Text block will be displayed as shown in Figure 8-30.

Figure 8-29: The Text block on the Complete Palette

Figure 8-30: The Text block

The Text block's Configuration Panel (shown in Figure 8-31) contains boxes for three pieces of text. In most cases, you'll fill in one or two of the boxes and supply the other input using a data wire. The output from the Text block is created by *concatenating* (or joining together) the three pieces of text. The Text block won't add a space between each item, so if you want a space between a label and a value, you'll need to add it yourself (you'll see an example of this in the next section).

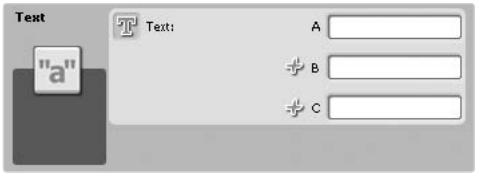


Figure 8-31: Configuration Panel for the Text block

adding labels to the displayed values

The previous version of the SoundMachine converted the Tone Frequency value to text and displayed the value. You can improve on this by using a Text block to add more information: Instead of just displaying a number, you'll display a label and the unit for the value. The unit used to measure frequency is hertz (meaning cycles per second), abbreviated as *Hz*. Instead of displaying `2500`, the program will display `Tone: 2500 Hz`.

Figure 8-32 and Figure 8-33 show the changes to the program and the Configuration Panel for the Text block. The Tone Frequency value is passed into the Text block as the B value, and the Configuration Panel is used to set the label and unit text. Although you can't see it in the Configuration Panel, there is a space after the label: The value entered is `Tone:` , not just `Tone:`. Likewise, there is a space before `Hz` in the bottom box.

The final change to this program adds the volume level to the display. The new code is similar to the code used to display the Tone Frequency value, taking the output from the Sound block's Volume plug and displaying it as `Volume: 50%`.

Figure 8-34 shows the changes to the program, and Figure 8-35 and Figure 8-36 show the Configuration Panels for the new Text and Display blocks. The Text block settings are similar to the ones used for displaying the tone frequency (for the volume there should be a space after the label but not one between the value and the percent sign). For the new Display block, in addition to setting the Action to Text, change the Line setting so that the volume is shown below the tone (instead of writing over it), and uncheck the Clear option so that the tone display is not erased.

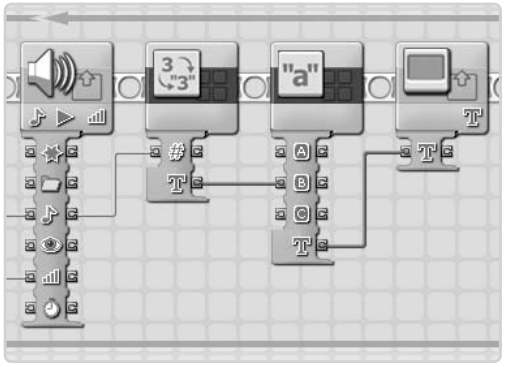


Figure 8-32: Text block added

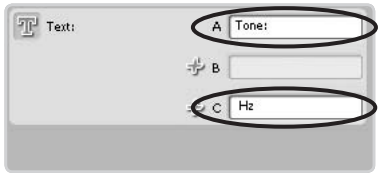


Figure 8-33: Configuration Panel for the Text block

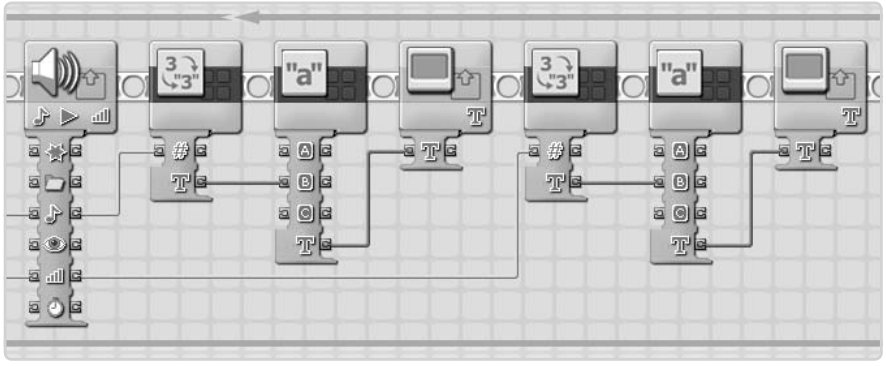


Figure 8-34: Displaying the volume

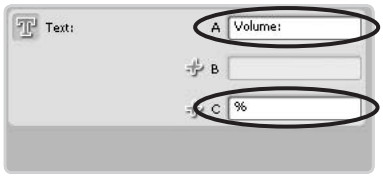


Figure 8-35: Configuration Panel for the Text block

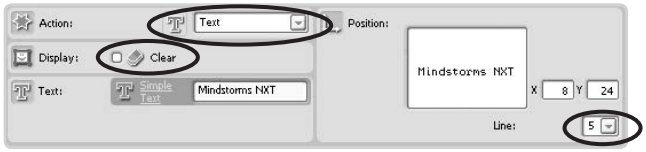


Figure 8-36: Configuration Panel for the Display block

When you run this version of the SoundMachine program, both the Tone and Volume values should be displayed. Notice that if you keep turning the volume dial, the display will show a value larger than 100 percent, even though the Sound block clamps the value to 100. This is because the value passed out of the Sound block is the same value that was passed in, not the value the block uses if the value is out of the normal range.

dealing with broken wires

As you are writing your program, it's possible to connect a data wire in such a way that it won't work correctly. A data wire that is not connected properly is known as a *broken wire*. A broken wire is shown as a dashed, gray line (as shown in Figure 8-37) to let you know that the program has a problem.

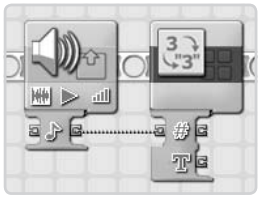


Figure 8-37: A broken wire

A data wire can appear broken for the following reasons:

- **The data types of the two plugs do not match.** For example, you can't connect the Tone Frequency output plug from the Sound block (a number) to the Display block's Text plug. Instead of connecting these two data plugs directly, you need to use the Number to Text block to change the value's data type.
- **There is no input data.** This happens when you connect a data wire to the output side of a pass-through plug without connecting the input side. This is the problem shown in Figure 8-37.
- **There are too many inputs.** Each input plug can be connected to only one output plug (otherwise it wouldn't know which value to use). An output plug can be connected to more than one input plug because there's no problem with passing the same value to multiple blocks.
- **The data wires form a cycle.** You can't connect one block to another that appears earlier in the program. In a simple program, the IDE won't allow you to draw a data wire that could cause a cycle. In programs using multiple Sequence Beams, it's possible to create a cycle by connecting blocks on different Sequence Beams, as you'll see in Chapter 17.

To determine the problem with a broken wire, select the wire, and look at the Help Panel (located in the lower-right corner of the IDE). A brief description of the problem should be displayed, as shown in Figure 8-38. Click More help to open the help file with a more complete explanation of the problem.

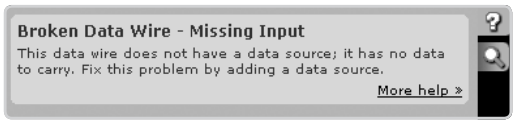


Figure 8-38: Help for a broken wire

If you encounter broken wires while editing a program, you don't necessarily have a problem; the wires may break as a result of the order in which you connect them to the blocks. For example, when building the SoundMachine program, if you first connect the Sound block to the Number to Text block, the wire will be broken. When you then connect the Math block to the Sound block, the broken wire will be repaired. Regardless, you must fix any broken wires before you run your program.

conclusion

Data wires move information between blocks, allowing you to change a block's settings while your program is running. The programs presented in this chapter have shown you some simple ways to use data wires and introduced some of the blocks that are designed to work with data wires. The Sensor blocks use data wires to make sensor readings available to other blocks in your program. The Math, Number to Text, and Text blocks are used almost exclusively with data wires to transform data or convert between data types.

The next two chapters will show you how to use data wires with the Switch and Loop blocks. You'll get lots of practice using data wires because you'll use them extensively for the programs in the remainder of this book.

9

data wires and the switch block

The Switch block is used to make decisions about which blocks to run by choosing between two or more alternatives. For example, the WallFollower program in Chapter 7 uses a Switch block that reads the Ultrasonic Sensor and decides whether to move the TriBot toward or away from the wall. In this chapter, you'll learn how to use the Switch block to make decisions using a value supplied by a data wire, instead of using data directly from a sensor. You'll also learn how to pass data between the blocks inside the Switch block and the ones that come before or after the Switch block.

the switch block's value option

The Control setting on the Switch block's Configuration Panel offers two choices: Sensor and Value. In the previous chapters, you used the Sensor option to make decisions based on input data supplied directly from a sensor. The Value option lets you make a decision using a value from a data wire.

When you select the Value option, the Configuration Panel should look like Figure 9-1, and an input data plug should appear at the bottom of the Switch block (as shown in Figure 9-2). This data plug is where you connect the data wire supplying the input value.

The Configuration Panel will look slightly different depending on the data type of the value you are using (Number, Text, or Logic). When you connect a data wire to the block's input plug, the correct type should be selected automatically, but you can select the data type for the Switch block's input plug manually from the Type list, once you have a data wire connected. Changing the Type setting will delete the data wire (otherwise there would be a data type mismatch error).

To connect a data wire to the Switch block, you draw the wire from an output plug to the Switch block. The IDE won't let you draw the wire starting from the Switch block's input plug.

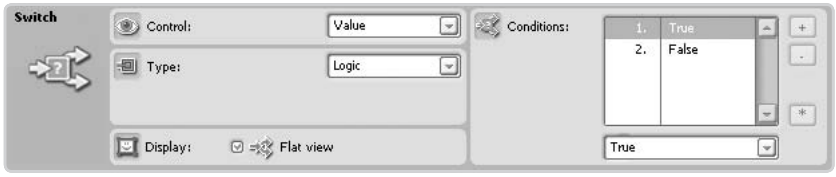


Figure 9-1: Switch block Configuration Panel using a Value control

Figure 9-2: The Switch block's input data plug

rewriting the GentleStop program

Recall that the GentleStop program makes the TriBot slow down and gently stop as it approaches a wall. The robot stops when the Power setting for the Move blocks is too low to make the robot move, but a better approach would be to stop the motors when the robot is close to the wall, instead of letting the motors stall. In this section, you'll rewrite the GentleStop program to do just that, using the output from the Ultrasonic Sensor block as the trigger for a Switch block. The program will work like the original, but you'll use a different approach to programming it.

You'll write the program in two parts. In the first part, you'll make the TriBot move forward and then stop when the reading from the Ultrasonic Sensor is less than 10. Unlike the original program, this version doesn't use the distance reading to control the motor's Power setting. The second part of the program will restore the connection between the Ultrasonic Sensor block and the Move block.

Here is the pseudocode for the first part of the program, which keeps the TriBot moving forward until it's 10 cm or less from the wall, at which point the motors are stopped:

```
begin loop
   read the value from the Ultrasonic Sensor
   if the distance > 10 then
      move forward
   else
      stop moving
   end if
loop forever
```

Figure 9-3 shows the program. The Ultrasonic Sensor block reads the sensor and compares the distance reading to the target value, which is set to 10. The result of this comparison is passed to the Switch block using a data wire connected to the Ultrasonic Sensor block's Yes/No output data plug. If the value is true, meaning the distance is greater than 10, the Switch block will execute the Move block on the upper Sequence Beam, moving the robot forward. If the distance is 10 or less, the block on the lower Sequence Beam will be used, and the robot will stop moving. Figures 9-4 through 9-6 show the Configuration Panels for the Ultrasonic Sensor and Move blocks.

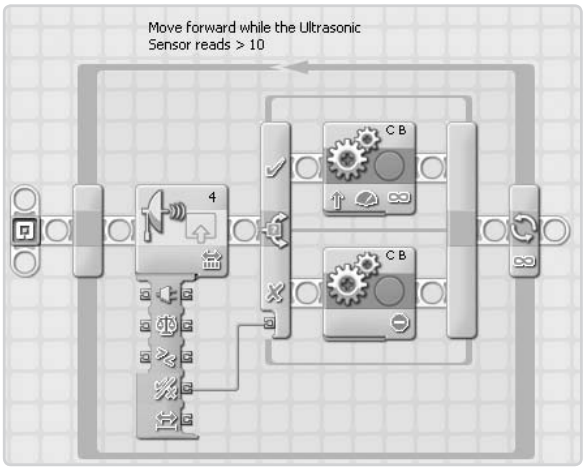


Figure 9-3: Moving forward while the distance is greater than 10 cm

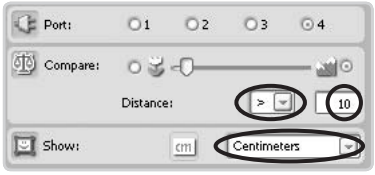


Figure 9-4: Configuration Panel for the Ultrasonic Sensor block

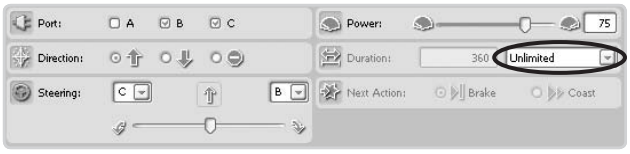


Figure 9-5: Moving forward

Figure 9-6: Stopping the motors

When you run this program, the TriBot should move forward quickly and then abruptly come to a stop when it gets close to a wall. Unlike the original GentleStop program, in this version of the program, the robot's speed doesn't depend on the distance from the wall.

advantages of using a sensor block

You could write the GentleStop program without using the Sensor block by using a Switch block configured to use the Ultrasonic Sensor and making the comparison with the trigger value (in this case 10). However, that approach wouldn't allow enough flexibility for the next step. Although having the Switch block work directly with a sensor is simple, you gain some advantages from using a Sensor block and passing the result of the comparison to the Switch block using a data wire:

- You can configure a Sensor block using data wires. For example, you can change the trigger value while the program is running. When using a Switch block, the settings for the sensor can be set using the Configuration Panel only.
- You may want to use a condition that's more complex than a simple greater than or less than comparison. Using data wires and the Math, Logic, and Comparison blocks, you can test for just about any condition you can think of.
- A Sensor block gives you access to the sensor reading as well as the result of the comparison. For example, in the next section, you'll use the Distance value to control the Move block's Power setting, in addition to using the Yes/No value to stop the robot.

passing data into a switch block

To make the TriBot's speed depend on its distance from the wall, the Distance value from the Ultrasonic Sensor needs to be connected to the Move block's Power plug, just as in the original program. To make a data wire cross the boundary of the Switch block, you must first unselect the Switch block's Flat view option (shown in Figure 9-7). Once the Switch block is using tabbed view, a data wire can be drawn between the Ultrasonic Sensor block's Distance plug and the Motor block's Power plug, as shown in Figure 9-8.

Figure 9-7: Unselecting the Flat view option

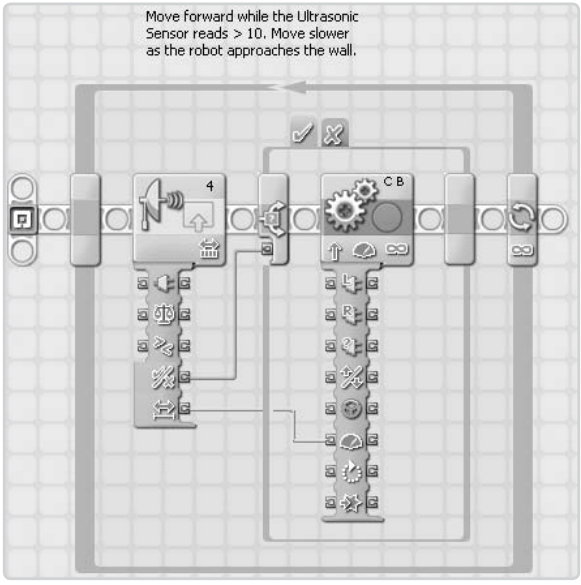


Figure 9-8: Connecting the Distance to the Power setting

When you run this version of the program, it should act like the original, except that the TriBot should stop the motors when it gets close to the wall, instead of letting them stall because the Power level is too low.

passing data out of a switch block

Passing data out of a Switch block can be a little more complex than passing data in because of the way the wires need to be connected. The LogicToText program shown in Figure 9-9 is a very simple example of passing data out of a Switch block. This program reads the Touch Sensor and writes `True` on the display if the button is pressed and `False`

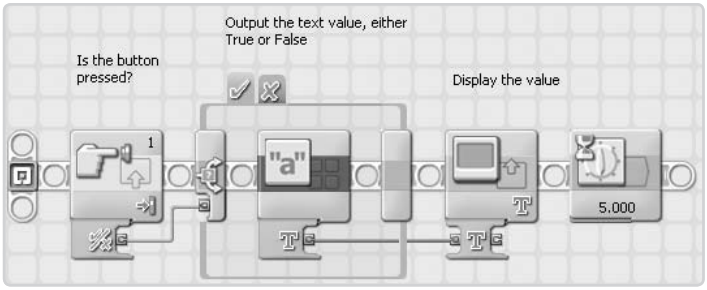


Figure 9-9: The LogicToText program

if it's not. Both Sequence Beams inside the Switch block contain a Text block whose output plug is connected to the Display block. The tricky thing here is that the Text blocks on both Sequence Beams need to be connected to the Display block.

I'll give step-by-step instructions to illustrate how to connect the data wires. Start with the program as shown in Figure 9-10. The Touch Sensor block uses the default values, which look for the button being pressed, and the result is used as the trigger value for the Switch block. The Text block on the Switch block's upper Sequence Beam generates the text `True`, and the one on the lower Sequence Beam generates `False`. To show the text, set the Display block's Action option to Text, since the default is Image. Figures 9-11 through 9-14 show the Configuration Panels for the Switch, Text, and Display blocks.

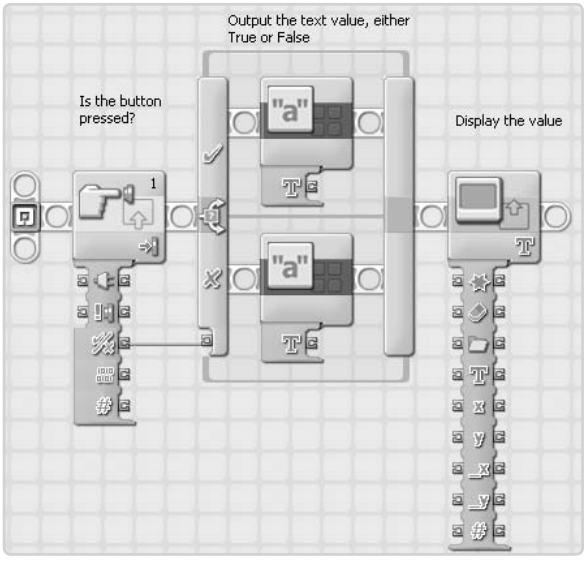


Figure 9-10: The LogicToText program starting point

Figure 9-11: Switch block Configuration Panel

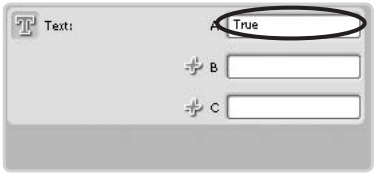


Figure 9-12: Text block on the upper Sequence Beam

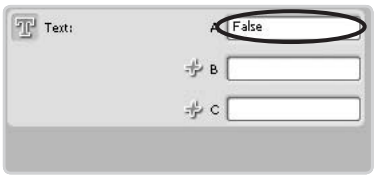


Figure 9-13: Text block on the lower Sequence Beam

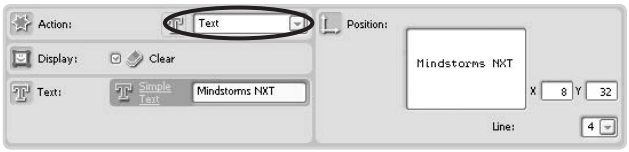
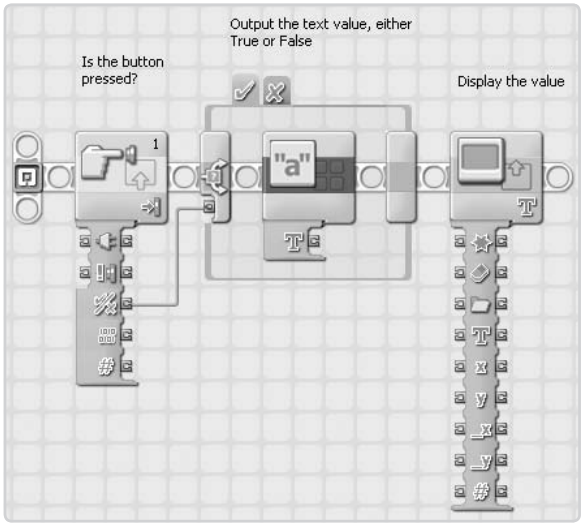


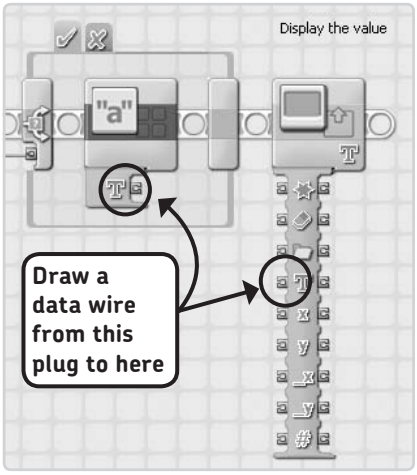
Figure 9-14: Configuration Panel for the Display block

Now comes the interesting part. Follow these steps to connect the output from the Text blocks to the Display block:

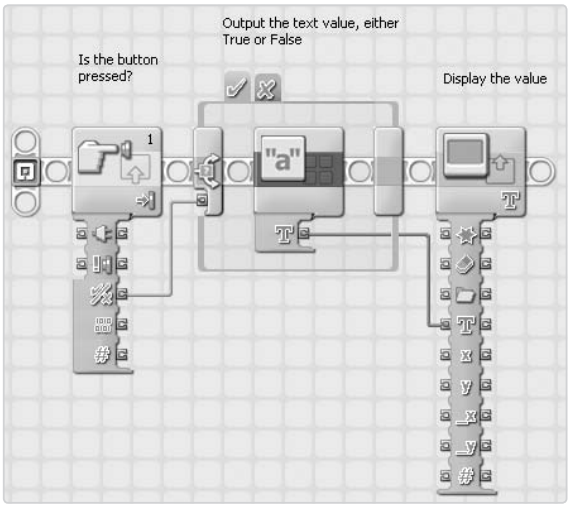
1. Uncheck the **Flat view** option of the Switch block. The program should look like this:



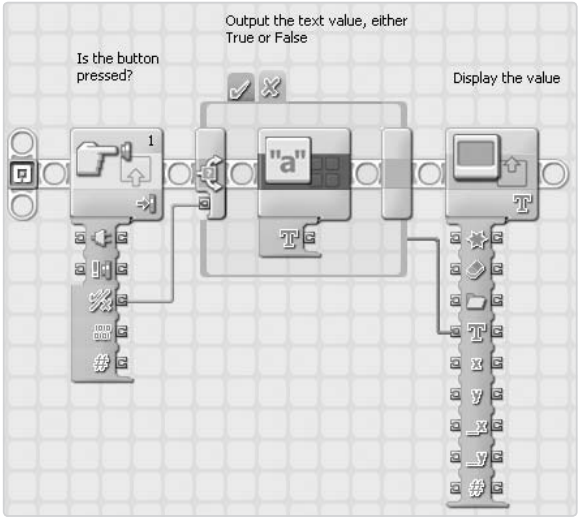
2. Draw a data wire from the Text block's Combined Text plug to the Display block's Text plug:



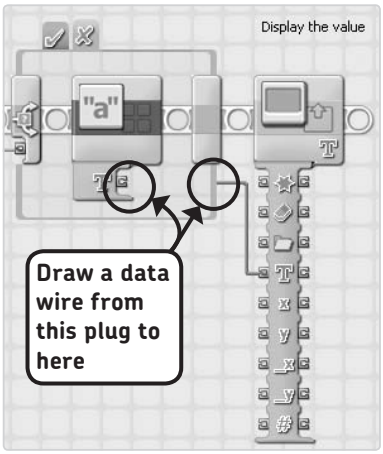
The program should now look like this:



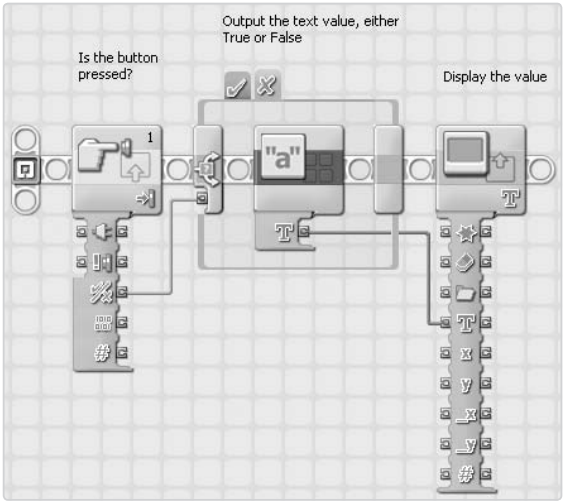
3. Click the X tab at the top of the Switch block to display the other Sequence Beam. The program should look like this:



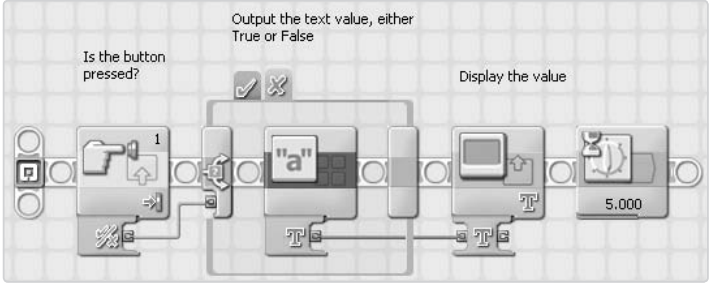
4. You can't connect the second Text block directly to the Text plug of the Display block, because an input plug can be connected to only one source. Instead, you need to connect the Text block's Combined Text plug to the edge of the Switch block at the point where the existing data wire crosses it, like this:



When the Switch block runs, only one of the tabs will be used (either the true one or the false one), and the output of the Text block will be sent to the Display block. The program should look like this:



5. Add a Wait Time block to pause the program so you can read the display before it's cleared. You can also shrink the data hubs to tidy up things. The completed program should look like this:



When you run the program, it should display either `True` or `False` depending on whether the Touch Sensor is pressed or released when the program starts. To test the true case, press and hold the Touch Sensor button before running the program, because the sensor is tested immediately upon starting the program.

matching more than two values

The Switch block in the LogicToText program uses the logic value from the Touch Sensor block to choose between two options. This works great with a logic value because it can have only two possible values (either true or false). But what if you want to choose from more than two options? If so, you'll need to use either a number value or a text value, since these data types can have more than two values.

When using a number or text value with a Switch block, you create a list of values to match against, one for each choice you want the program to make. Figure 9-15 shows the Configuration Panel for a Switch block that has been configured to choose between three numbers: 10, 25, and 75.

Figure 9-15: Configuration Panel with three conditions

In the center of the Conditions section is a numbered list of the values to match against. In the Work Area, the Switch block will show a tab for each option. In this example, the Switch block has three choices and therefore will show three tabs along the top (shown in Figure 9-16). The tabs are in the same order as the items in the list, so the first tab will be used if the input is 10, the middle tab for 25, and the third tab for 75.

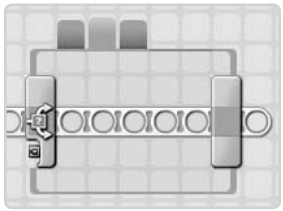


Figure 9-16: A Switch block with three choices

The box at the bottom of the list is used to edit the value for the selected item. The IDE usually keeps the list of conditions in order (either numerical or alphabetical), so changing the value will also change an item's position in the list. For example, if you change the second item's value from 25 to 125, it will move to the bottom of the list, as shown in Figure 9-17. This choice now corresponds to the third tab of the Switch block, instead of the second. The blocks within the Switch block will be moved to the correct tabs automatically; the only things that change are the order of items in the list and the order of the tabs on the Switch block.

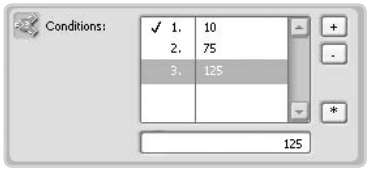


Figure 9-17: After changing 25 to 125, the item has moved.

adding and removing conditions

The [+] button adds a new condition to the end of the list. To use more than two choices, first uncheck the Flat view option. When the Flat view option is selected, you can set the values for the two default conditions, but you can't add more conditions.

The [-] button removes the select item from the list. A Switch block must have at least two choices, so this button will be disabled if there are fewer than three items in the list.

the default condition

What happens if you pass 35 to this Switch block? This isn't one of the choices (10, 25, and 75), but the Switch block has to choose one of the three tabs. The check mark next to the top condition in Figure 9-17 indicates that this is the default choice, meaning that any value other than 10, 25, or 75 will also match this condition. So, the first tab will be used if you pass 35 to the Switch block.

You can choose which value should be used as the default by selecting an item in the list with the mouse and then clicking the [*] button.

If you remove the item selected as the default, then the first item in the list becomes the default.

NOTE Sometimes when you delete the default item, the IDE will incorrectly show the check mark next to the last item or not show it at all. Selecting a different block and then reselecting the Switch block will put the check mark next to the correct item.

using numbers with the NXT-G 2.0 switch block

One of the major differences between NXT-G 1.1 and NXT-G 2.0 is the change from using only whole numbers to using floating-point numbers, and this affects how the Switch block works. Even though NXT-G 2.0 supports floating-point numbers, you can use only whole numbers for the items in the Conditions list. The Switch block will round the value on the input data wire to the nearest whole number before comparing it with the list of values. For example, using the values shown earlier in Figure 9-16 (10, 75, and 125), any value greater than or equal to 74.5 and less than 75.5 will be rounded to 75 and so will match the middle choice.

fixing the SoundMachine program's volume display

The SoundMachine program from Chapter 8 allows you to control the Volume and Tone of a Sound block using the TriBot's two wheels, but the way the program displays the Volume level is not quite correct. Although the Sound block accepts values in the range of 0 to 100 for the Volume, the block really supports only five volume settings. In effect, the input value is rounded down to 0, 25, 50, 75, or 100.

In this section, you'll improve the program by displaying a description for the Volume instead of a number. To do so, you'll use a Switch block with six choices to convert the numeric Volume setting to a text description.

The first step is to change the Volume to one of six possible values to match the way the Sound block works. By doing a little math, you can convert the Volume setting to a number between 0 and 5 and use this value as the input into a Switch block. Table 9-1 shows Switch block input values and the descriptive text to use for each Volume level. A Volume setting greater than 124 will generate a result of

5 or greater, and the program will display --- to indicate an out-of-range condition.

table 9-1: volume levels and descriptions

volume range	switch block input value	description
0–24	0	Off
25–49	1	Low
50–74	2	Medium
75–99	3	Loud
100–124	4	Loudest
Greater than 124	5 or greater	---

The process of converting the Volume setting to the Switch block input value depends on which version of NXT-G you're using. The following two sections describe the process for each software version.

calculating the input value using NXT-G 1.1

With NXT-G 1.1, all the numeric values are whole numbers, and the result of any division operation is truncated, meaning the fractional part is dropped. For example, if the volume setting is 70 and you divide this by 25, you'll get 2 instead of 2.8. This actually makes the conversion easy; all you need to do is divide the Volume setting by 25 to get the correct Switch block input value shown in Table 9-1.

calculating the input value using NXT-G 2.0

When using NXT-G 2.0, the division operation will keep the fractional part. For example, dividing a Volume setting of 70 by 25 will give a result of 2.8. If you pass this value to the Switch block, it will be rounded to 3. However, according to Table 9-1 (which matches the way the Sound block works), a Volume setting of 70 should correspond to an input value of 2. The real issue here is that the Sound block truncates the value, but the Switch block rounds the value. This is a common problem when working with floating-point numbers and is easy to fix once you're aware of it. All you need to do is subtract 0.5 from the value before passing it to the Switch block. This will cause the rounding operation to have the same effect as truncating the original value.

modifying the program

To use a text description for the Volume, the Number to Text block circled in Figure 9-18 must be replaced with a Math block (two Math blocks for NXT-G 2.0) and a Switch block. The Switch block will have six conditions, each of which uses a Text block to put the appropriate description on a data wire. The data wires from the Text blocks within the Switch block will connect to the Display block, just like in the LogicToText program.

Follow these steps to make the necessary changes to the program:

1. Open the SoundMachine program.
2. Select the Number to Text block indicated in Figure 9-18, and use the DELETE key to remove it from the program.

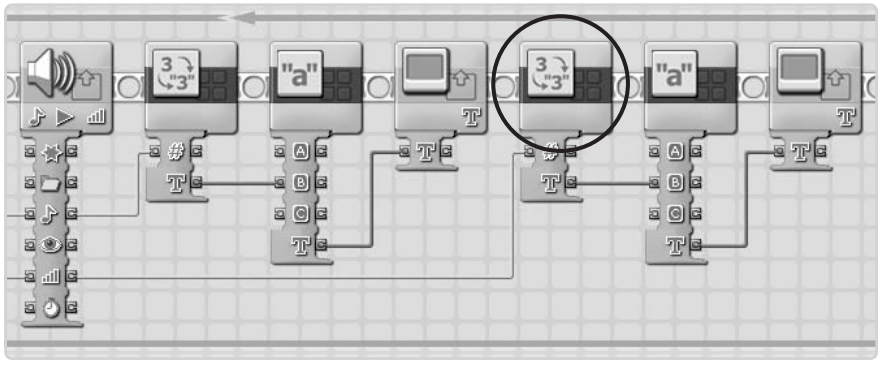
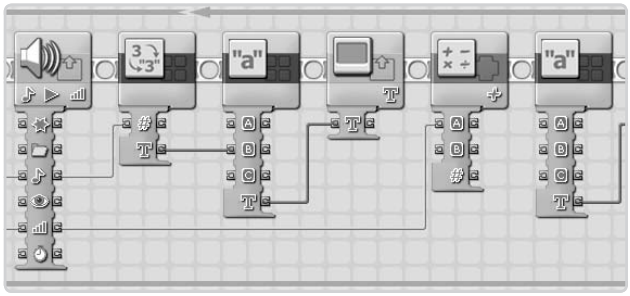
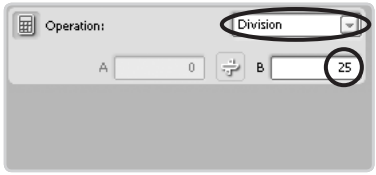


Figure 9-18: Replace the Number to Text block with Math and Switch blocks

3. Add a Math block just after the first Display block. Connect the Sound block's Volume output plug to the Math block's A input plug. The program should look like this:

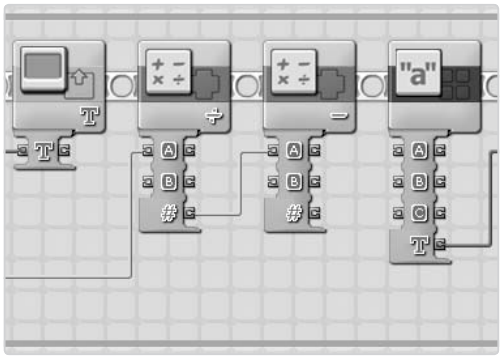


4. In the Math block's Configuration Panel, set the Operation value to **Division**, and set the B value to **25**.

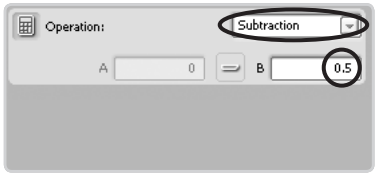


5. If you're using NXT-G 2.0, add another Math block and connect the output plug of the first Math block to this block's A input plug. This part of the program should look like this:

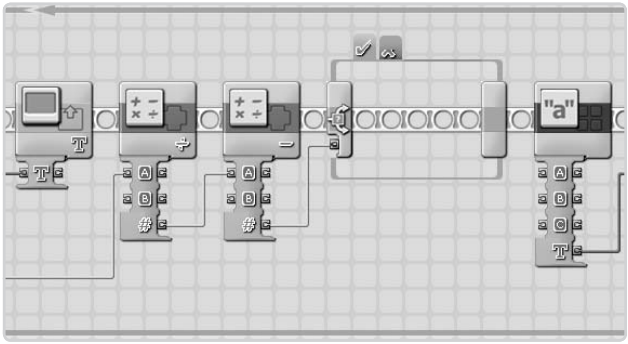
NXT 2.0



6. Set the Math block's Operation option to **Subtraction**, and set the B value to **0.5**. (This step is only for NXT-G 2.0 users.)

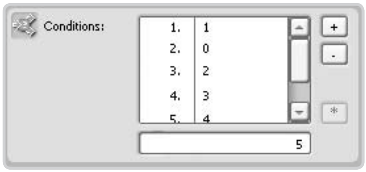


7. Add a Switch block after the Math block, then uncheck the **Flat view** option, and set the Control option to **Value**. Now connect the output of the Math block to the Switch block's input plug. The images that follow show how the program looks using NXT-G 2.0; if you're using NXT-G 1.1, you should have only one Math block. The modified section of the program should look like this:



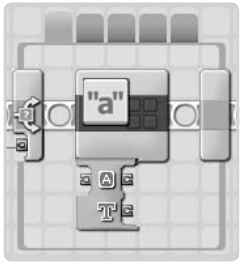
8. The Switch block's Configuration Panel will have two choices by default (0 and 1). Add four more choices for the numbers 2 through 5. When you add a new choice, the IDE will supply a value larger than any of the existing choices, so for this program you don't need to change any of the values; the defaults just happen to be correct.
9. Select the last choice (**5**), and click the ***** button to set it as the default choice, because it's the out-of-range value, and any value larger than 5 is out of range.

At this point, the Switch block has all the numbers you want to use, but it may not have them in the correct order. Usually the list of numbers is kept in order, but the first two (0 and 1) may be switched, as shown here:

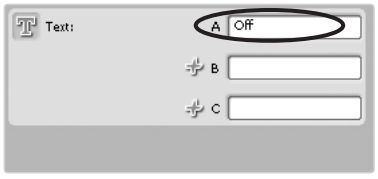


I like to keep the items in order so I don't accidentally use the wrong tab. To get the items back in order, select the second item (0), and change it to 10, which puts it at the bottom of the list. Then change the 10 to 0; it will move to the top, and the list will be back in order.

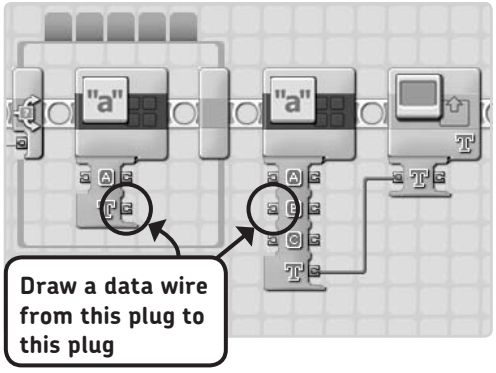
10. The Switch block should now show five tabs across the top. Select the first tab, and drag a Text block onto the Sequence Beam.



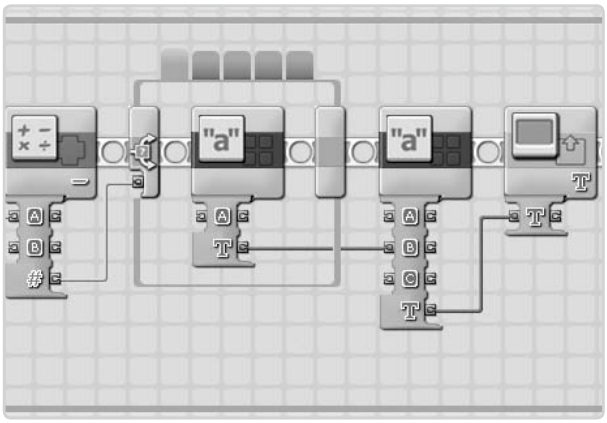
11. The block on the first tab will be used when the volume is off. Enter **Off** in the box at the top of the Text block's Configuration Panel:



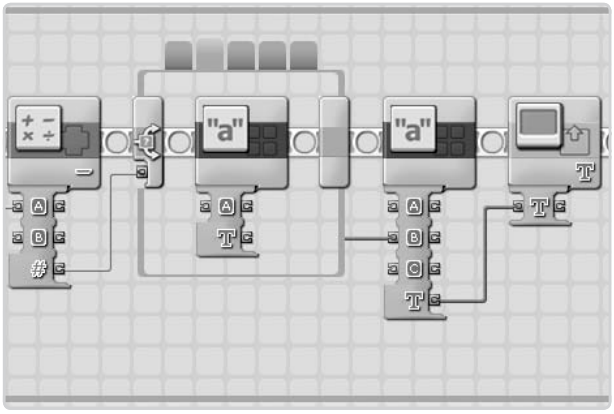
12. Connect the Text block's Combined Text data plug to the B data plug to the Text block to the right of the Switch block, as shown here:



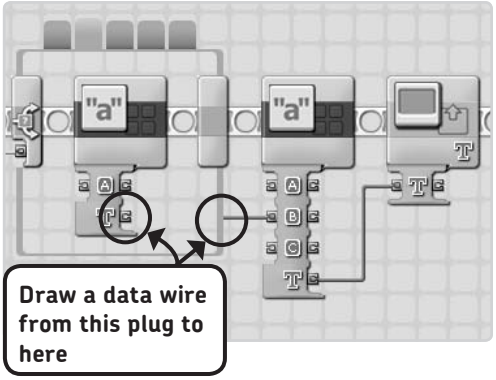
This section of the program should now look like this:



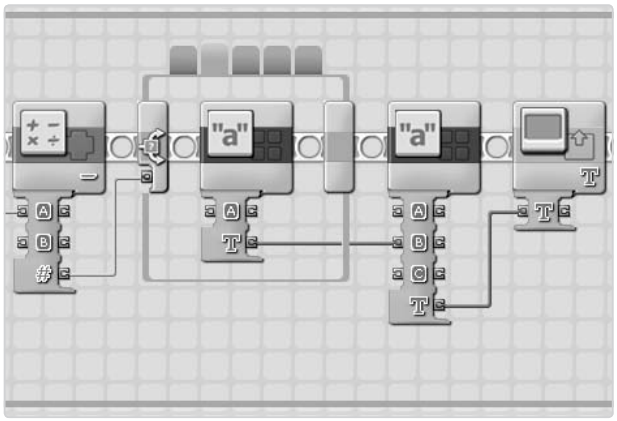
13. Select the Switch block's second tab, and add a Text block. Set the Text option to **Low**. This section of the program should now look like this:



14. Draw a data wire from the Text block's Combined Text output plug to the edge of the Switch block to meet up with the data wire connected to the Text block, as shown here:



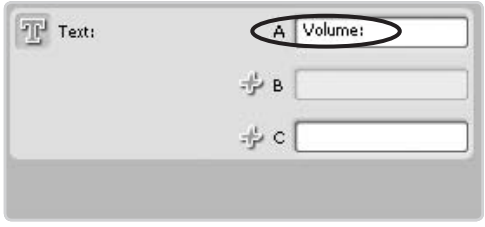
This section of the program should now look like this:



15. Repeat the same steps to add a Text block to the remaining three tabs of the Switch block. Set the Text option for each block using the appropriate values from Table 9-1, and connect the output from each Text block to the data wire on the side of the Switch block.
16. The Switch block has six choices, but only five tabs are displayed along the top. When you select an item in the Conditions list, the tab for that item is shown in the Switch block. Select the sixth choice in the Switch block's Configuration Panel, and add the final Text block. Set the Text option to **---**, and connect the data wire as with the previous Text blocks.

NOTE **The Switch block will show as many tabs as can fit along the top. If more than one block is dropped into the Switch block, the block will grow wider, and more tabs will be displayed.**

17. Select the Text block to the right of the Switch block, and then remove the percent sign from the bottom box. The Configuration Panel should look like this:



Now when you run the program, it should display a description of the volume level. If the Volume description is sometimes blank, then the most likely source of the problem is that the data wire on the inside of the Switch block doesn't meet the data wire on the outside of the Switch block. If this happens, use these steps to resolve the problem:

1. Select the Switch block tab corresponding to the missing text.
2. Click the data wire between the Text block and the edge of the Switch block. Make sure that only the data wire is selected; if the Configuration Panel is still displayed, then one of the blocks is selected and not the data wire, and you should click an empty spot in the Work Area to unselect all the blocks and then click the data wire again.
3. Press the DELETE key to remove the data wire.
4. Draw a new data wire between the Text block and the edge of the Switch block.

Once the new data wire is in place, you should test the program again. It's not easy to see whether the data wires meet at the edge of the Switch block at just the right spot, so it may take a few tries to get it right.

conclusion

Using a data wire to supply the input to a Switch block gives you a lot of flexibility in the types of decisions your programs can make. Using a Sensor block, you can perform the comparison outside the Switch block, allowing you to make more complex decisions than the Switch block alone supports.

Using a number or text value as input, you can have the Switch block choose from more than two possible alternatives. Passing data between the blocks within the Switch block and those before or after allows you to easily customize each alternative action or make decisions that affect the rest of the program.

10

data wires and the loop block

In this chapter, you'll learn how to use two special features of the Loop block that are designed to be used with data wires. The Loop block has two data plugs: The Loop Count data plug tells you how many times the loop has completed, and the Loop Condition data plug allows you to control when the Loop block finishes. I'll begin with a few simple programs to demonstrate how these features work, and then I'll show you how to use them to build three different programmable timers.

the loop count

The loop count tracks the number of times the loop body has been repeated. The data plug for the count isn't displayed when you first add the Loop block to your program, but if you check the Show Counter box in the Loop block's Configuration Panel (Figure 10-1), it will appear on the Loop block, as shown in Figure 10-2.

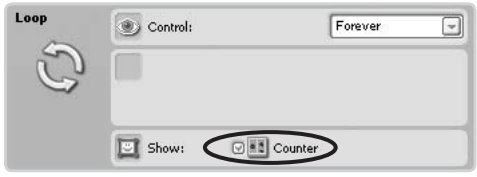


Figure 10-1: Check the box to display the Loop Count data plug.

Figure 10-2: The Loop Count data plug

The value written to the Loop Count data plug equals the number of times the loop has completed, and a new value is written each time the loop repeats. The value is put on the data wire before the loop body executes, so the first time through, the loop the value is 0, the second time through the value is 1, and so on. When the loop finishes, the value on the data wire will be one fewer than the number of times the loop completed because it isn't updated after the last time the body is run.

creating the LoopCountTest program

The LoopCountTest program (shown in Figure 10-3) demonstrates how the loop count behaves. In this program, the Loop block repeats five times, and each time through the loop count is displayed. The Wait Time block adds a short pause to give you time to read the value. Figure 10-4 shows the Loop block's Configuration Panel.

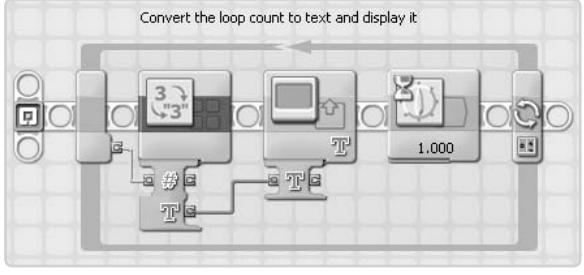


Figure 10-3: The LoopCountTest program

When you run this program, the display should show 0, 1, 2, 3, and 4.

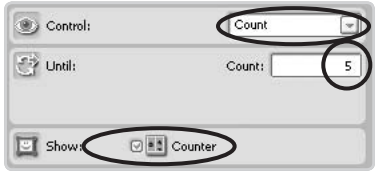


Figure 10-4: The Configuration Panel for the Loop block

NOTE **This program and the two that follow aren't very interesting to watch (you won't impress your friends by building a robot that counts to four), but they're not designed to be impressive. They're here to demonstrate how the NXT-G language works. When you begin working with a block or feature you haven't used before, it can be helpful to write small programs like these to gain a more complete understanding of the language. That will allow you to write programs that really are interesting.**

restarting a loop

Some programs use nested Loop blocks, with one loop inside another. The LoopCountTest2 program shown in Figure 10-5 is a simple example of this. The program nests the code from the LoopCountTest program in another Loop block. The outer Loop block is configured to run twice, as shown in Figure 10-6.

The first time the inner loop is run, the display should show 0, 1, 2, 3, and 4 just like LoopCountTest. But what will happen the second time the inner loop runs? There are two reasonable answers: It could display 0, 1, 2, 3, and 4 again or continue counting and show 5, 6, 7, 8, and 9.

In fact, when you run the program, the display will show 0, 1, 2, 3, 4 and then repeat 0, 1, 2, 3, 4. This tells you that the loop count is reset to 0 each time the Loop block is started.

setting the final loop count value

The LoopCountTest and LoopCountTest2 programs use the loop count within the loop body. This same value can also be used by blocks that follow the Loop block, as demonstrated by the LoopCountTest3 program (shown in Figure 10-7). LoopCountTest3 prints the final loop count value on the display. The Loop block is configured to run five times, as shown in Figure 10-8.

Because the loop count value that the program displays is put on the data wire at the start of the last run through the loop, it will be one fewer than the total number of times the loop repeats. This means that when you run the program, it should print 4 on the display, because the Loop block is set to run five times.

NOTE **It's common in computer programming to begin counting from zero instead of one. This doesn't usually pose a big problem, but it's easy to get *off-by-one errors*, where a loop repeats one too few or one too many times.**

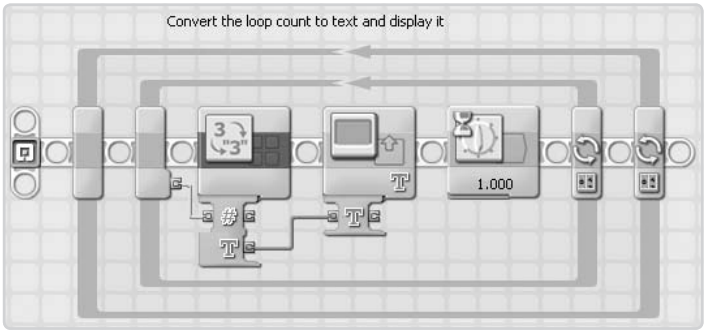


Figure 10-5: The LoopCountTest2 program

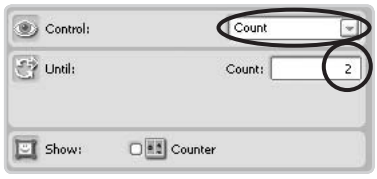


Figure 10-6: Configuration Panel for the outer Loop block

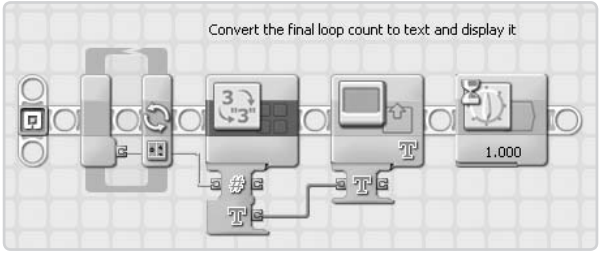


Figure 10-7: The LoopCountTest3 program

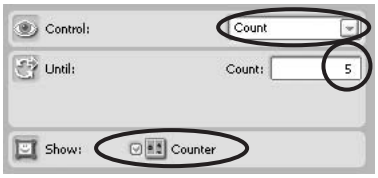


Figure 10-8: Looping five times

setting the loop condition

The Loop block uses the loop condition to decide when to exit the loop. You can set the loop condition using a data wire in the same way that you use a logic value with a Switch block.

Like the Loop Count data plug, the Loop Condition data plug is not shown by default; to make it visible, set the Loop block's Control value to Logic (as shown in Figure 10-9), and the Loop Condition data plug should appear on the right side of the Loop block, as shown in Figure 10-10.

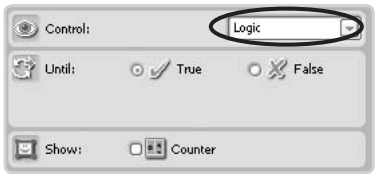


Figure 10-9: Set the Control value to **Logic**.

Figure 10-10: The Loop Condition data plug

The choice you make in the Until section of the Configuration Panel determines which value (true or false) exits the loop. The value is supplied by connecting a data wire with a logic value to the Loop Condition plug.

The programs that follow use the Loop Condition plug to build three versions of a simple programmable timer. After experimenting with the code we develop here, you can reuse it in your own programs to control how long a program waits. However, before we develop the programs, let's take a look at the NXT timers and the Timer block.

timers

The NXT has three built-in timers that act like stopwatches. You can use a timer to tell you how long your program has been running or to measure how long it takes the robot to perform a particular task. Typically using a timer is a two-step process: You reset the timer to 0 before beginning a task, and then read the timer when the task is complete. Think of a timer as a kind of sensor for time.

Like a stopwatch, a timer can be used for a variety of purposes. For example, you can do any of the following:

- Time how long it takes your entire program to run and use that information to fine-tune parts of your program. For example, you might measure how long it takes your robot to solve a maze and then use that information to find the fastest solution.
- Time parts of your program to see whether you can speed up certain sections.
- Use timers to make your program perform a periodic action. For example, as part of an experiment, you could use a timer to read a sensor every 10 seconds over a period of 5 minutes.
- Use a timer to limit how long you wait for a sensor to reach an expected target value. This technique can help you avoid situations where your program stops working completely if something unexpected happens.

Because the NXT has three timers, you can perform several timing tasks, such as those listed previously, within the same program.

the timer block

Timers are used like the other sensors and can be selected from the sensor list of the Wait, Switch, and Loop blocks. You can also control a timer using the Timer block, which appears in the sensor group on the Complete Palette (as shown in Figure 10-11). In your program, the Timer block will appear as shown in Figure 10-12.

Figure 10-11: The Timer block on the Complete Palette

Figure 10-12: The Timer block

The Timer block's Configuration Panel (shown in Figure 10-13) allows you to either reset a timer to 0 or read the current timer value, by selecting the appropriate choice for the Action item. When you select Read, you can use the items in the Compare section to test the current timer value against a target value, or you can use a data wire to pass the value to other blocks in your program. The Timer item at the top of the Configuration Panel selects the timer (1, 2, or 3) to use.

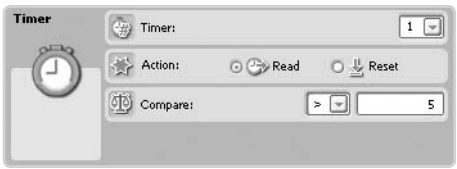


Figure 10-13: The Timer block's Configuration Panel

The trigger value that you enter in the Compare section is in seconds. For example, using the settings shown in Figure 10-13, the comparison will be true when the timer has been running for more than five seconds. We usually think of a second as a very small amount of time, but in the world of computers and moving robots, a second can be a very long time. For this reason, NXT-G actually tracks time in milliseconds, or thousandths of a second. However, in the interest of usability, the Configuration Panels use seconds for time values, with up to three decimal places. (NXT-G simply converts the value you enter into milliseconds.)

This is important because when you use time values with data wires, you *always* must use milliseconds. For example, you can set the trigger value to 5 seconds by entering 5 in the box on the Configuration Panel, but to set the same value using a data wire, you have to use 5,000 milliseconds (equal to 5 seconds).

NOTE **The help file for the Timer block shows the possible range for the Trigger Point data plug as 0 to 100, which could make you think the value is in seconds. As stated earlier, the value is actually in milliseconds, and the range should be given as 0 to 2,147,483,647.**

a programmable timer, version 1

Many of the programs presented so far have used the Wait Time block to add pauses. Although you can set the length of a pause using the Wait Time block's Configuration Panel, you can't set it using a data wire, which means that you can't change the length of time the block waits while a program is running. The next program shows how to get around this limitation by combining a Loop block with a data wire to control when the loop exits, thereby allowing you to adjust the delay programmatically.

The Timer1 program (shown in Figure 10-14) uses the Timer block and a Loop block to create a programmable timer. A Math block is used to put the length of the delay on a data wire, although in your own programs you would usually calculate the value based on certain input. In this example, the program expects the delay to be in milliseconds.

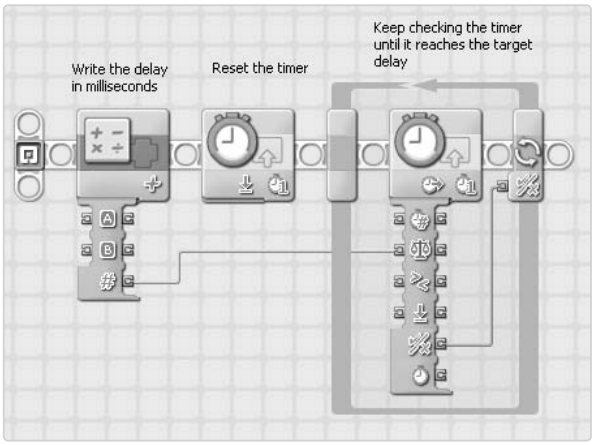


Figure 10-14: The Timer1 program

As you can see, the first Timer block resets the timer. Although this seems unnecessary in this small example (since the timers are all reset to 0 when the program starts), you'll need to reset the timer if you use this code within a larger program.

The Loop block keeps running until the timer reaches the trigger value passed to the second Timer block on the data wire. This is the real key to this program: The ability to use a data wire to set the Timer block's trigger value allows you to decide how long to pause while the program is running.

Figures 10-15 through 10-17 show the Configuration Panels for the Math block, the first Timer block, and the Loop block. (The second Timer block uses the default settings in the Configuration Panel because the only setting that needs to change is the target. It will be set using the data wire.)

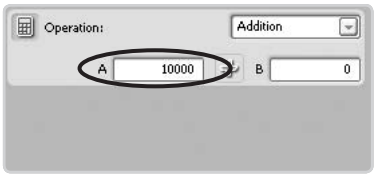


Figure 10-15: Writing 10000 (10 seconds) to the data wire used for the timer's target

Figure 10-16: Resetting the timer

Figure 10-17: Looping until the input value is true

When you run this program, the display screen should show that the program runs for 10 seconds and then stops. Although the program is not very compelling on its own, a programmable timer can be an important part of a larger program.

the compare block

The next programmable timer uses the Compare block to compare two numbers. The Compare block is in the Data group of the Complete Palette (shown in Figure 10-18) and will appear in your program as shown in Figure 10-19. You can supply the two input values using data wires or the Configuration Panel (shown in Figure 10-20). The comparison is made based on the operation selected, which can be either Less than, Greater than, or Equals. The result of the comparison is written to an output data plug.

Figure 10-18: The Compare block on the Complete Palette

Figure 10-19: The Compare block

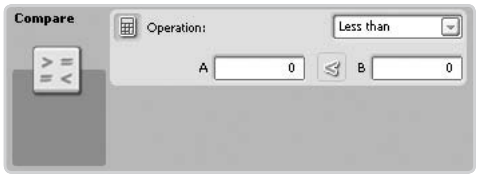


Figure 10-20: The Compare block's Configuration Panel

In some ways, the Compare block is like the Math block. Both blocks take two numbers as input and generate an output value. You select the operation on each using the Configuration Panel and supply the two numbers using either the Configuration Panel or data wires. However, the result from the Math block is a number, while the result from the Compare block is a logic value (either true or false). When using the Compare block, you're really asking a question like "Is 11 greater than 3?" The answer will be logic value, which in this case is true.

The Compare block is useful for making decisions because its result, a logic value, can be used to control both the Switch and Loop blocks. The Switch and Loop blocks work great when you just want to read a sensor and compare the reading with a target value. But sometimes you may want to do something a little more complicated. The Compare block allows you to compare any two numbers, giving you more flexibility than using a Switch or Loop block alone. For example, you can compare the readings from two Rotation Sensors, or you can use a Math block to modify a sensor value before comparing it with a target value. You can then use the result of the comparison to control a Switch or Loop block.

Typically, the Compare block is used as shown in Figure 10-21. Here data wires give the block two numbers to compare. In this example, the Operation option for the block is set to Less than (indicated by the less-than symbol in the lower-right corner of the block).

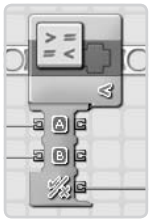


Figure 10-21: Comparing block with connections

When the block runs, it takes the two input numbers, compares them, and puts the resulting logic value on the output data wire. For example, if the A value is 7 and the B value is 12, the result will be true, because 7 is less than 12. On the other hand, if A is 25 and B is 8, the result will be false, because 25 is not less than 8.

a programmable timer, version 2

There is often more than one way to solve a problem, with each solution having its own advantages and disadvantages. One disadvantage of the Timer1 program is that it uses one of the three timers, so you'll need to use a different approach if your program uses all three timers for other tasks. The

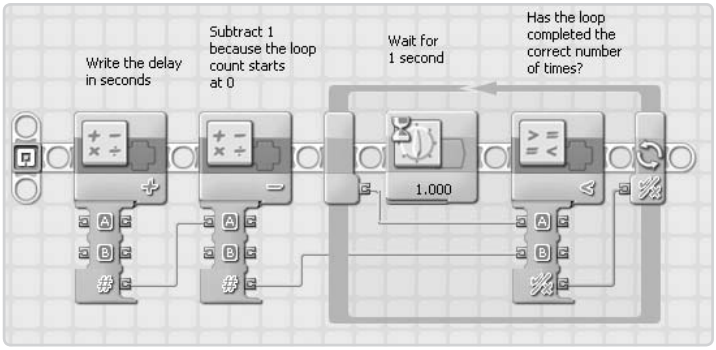


Figure 10-22: The Timer2 program

Timer2 program presented in this section demonstrates an alternative way to build a programmable timer.

Using data wires and a Compare block, you can control the number of times a Loop block repeats. By putting a Wait Time block inside a Loop block, you can control the length of time the program waits, as shown in the Timer2 program in Figure 10-22.

Like Timer1, Timer2 uses a Math block to supply the delay value. Inside the loop, the Wait Time block pauses for 1 second, so to pause for 10 seconds, you need to make the loop repeat 10 times. You can control how many times the loop repeats by comparing the loop count with the target value. (Recall that the loop count starts at zero, so you need to subtract one from the target value before doing the comparison, or the pause will be one second too long.) As long as the loop count is less than the target value, the loop repeats; when the loop count reaches the target value, the loop exits.

Listing 1 shows the pseudocode for this program. Notice that the loop continues as long as the result of the Compare block is true. The first time through the loop, the loop count is 0, and the target value is 9. Because 0 is less than 9, the result from the Compare block will be true. The loop will keep repeating until the loop count reaches 9, which will make the result of the Compare block be false (because 9 is not less than 9). To make the Loop block exit when the value from the Compare block is false, we need to change the block's Until setting to False, as shown in Figure 10-25. Figures 10-23 through 10-27 show the Configuration Panels for all the blocks.

```
use a Math block to write the delay in seconds to a
   data wire
subtract 1 from the delay
begin loop
   wait for 1 second
   is the Loop Count < the delay?
loop until the comparison is false
```

Listing 10-1: The Timer2 program

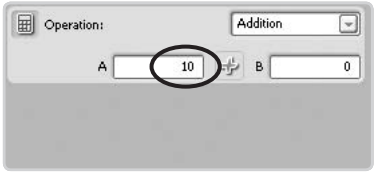


Figure 10-23: The target delay in seconds

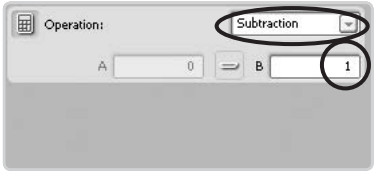


Figure 10-24: Subtracting one from the target value

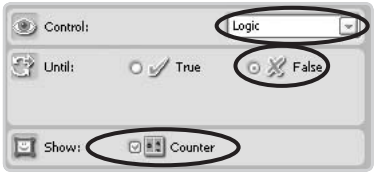


Figure 10-25: Looping until the comparison is false

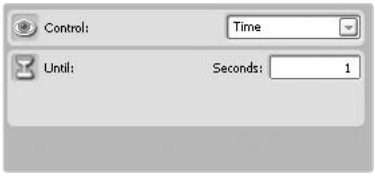


Figure 10-26: Waiting for one second

Figure 10-27: Is the Loop Count less than the target value?

This program should behave just like Timer1: The display should show that the program is running for 10 seconds and then show that it's done. Now a couple of items of note.

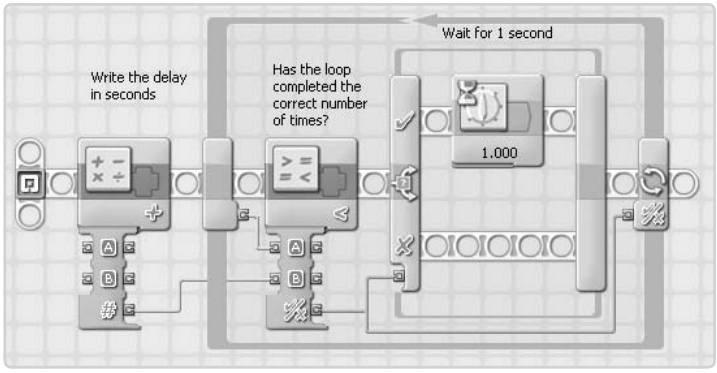


Figure 10-28: The Timer3 program

First, notice that the Timer2 delay is set in seconds, whereas Timer1 uses milliseconds. To change the delay to milliseconds, simply change the setting for the Wait Time block to 0.001 instead of 1. The unit you use for the delay (seconds, milliseconds, minutes, and so on) only needs to match the setting of the Wait Time block; the rest of the program can stay the same.

Second, the key to this program is combining the Compare block and the loop count to control the number of times the Loop block repeats. The usefulness of this technique isn't limited to building a programmable timer. You can use a similar arrangement of blocks in any situation where you want to control the number of times the loop repeats while the program is running.

a programmable timer, version 3

The Timer2 program has one potential problem: Because the Loop block checks the condition at the end of the loop, the timer will always wait at least one second. But what if the program determines that the delay should be zero or a negative number? The Timer2 program will still wait one second, which may not be the behavior you want.

To avoid pausing for one second when the delay is zero (or less), we need to check the loop condition at the start of the loop instead of at the end, as shown in the Timer3 program in Figure 10-28. In this program, the Compare block has been moved to the beginning of the loop, and the result is used to both exit the loop and control a Switch block. The Switch block makes sure that the Wait Time block is used only if the comparison is true, thus avoiding the delay if the starting value is 0 or less.

Figure 10-29 shows the Switch block's Configuration Panel. Notice that because the comparison is now done at the beginning of the loop, you don't need to subtract one from the delay before starting the loop.

Figure 10-29: The Switch block's Configuration Panel

Like Timer2, Timer3 will run for 10 seconds and then finish. If you change the delay to 0 (by changing the value in the first Math block), Timer 3 will finish immediately, instead of pausing for one second. You can use a similar approach any time you want to check the loop condition at the beginning of the loop instead of at the end.

conclusion

The Loop block can use data wires for both the loop count and the loop condition. Once you get used to the loop count starting at zero, it's easy to use it to control the loop. The Timer1 program demonstrates how to use the loop condition to control when the loop finishes. The Timer2 program shows how to use the loop count and a Compare block to control how many times the loop repeats. Finally, the Timer3 program shows how to rearrange the blocks within the loop when you want the loop condition checked at the beginning of the loop. This is useful in situations where you may not want the loop body to be executed at all.

11

variables

The previous three chapters explored using data wires to move data between blocks in your program. Although this ability can be very useful, for some problems you need to do more than just move data around. Often you'll need to tuck the information away and use it later in your program.

When you need to store information for later use, use a *variable*. In this chapter, I'll show you how to use variables and the types of problems they can help you solve.

a place for your data

Think of a variable as a place in the NXT's memory where you can store a value. The programs in this chapter will show you several ways that you can use variables in your programs.

The first example program in this chapter is a modified version of the RedOrBlue program from Chapter 5 (shown in Figure 11-1). Recall that this program uses the Color or Light Sensor to tell whether an object is red or blue. The version that you'll develop in this chapter will use variables to count the number of red and blue objects and show the running totals on the screen.

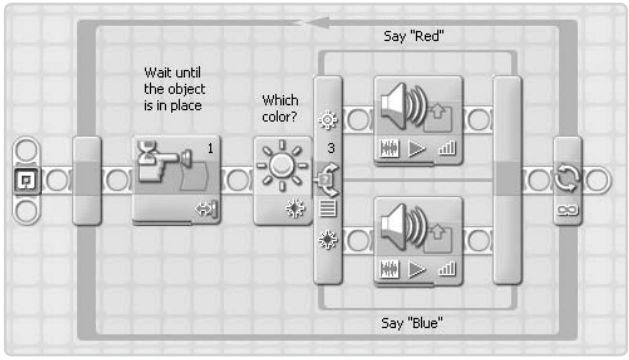


Figure 11-1: The RedOrBlue program

managing variables

Using variables in NXT-G is a two-step process. First you create the variable, and then you use the Variable block in your program to work with the data contained in the variable.

To create a variable, open the Edit Variables dialog (shown in Figure 11-2) using the Edit ▸ Define Variables menu item. The top half of the dialog lists all the variables defined for your program.

To create a new variable, click the Create button, and a new variable will be created with a name like *Variable 1*.

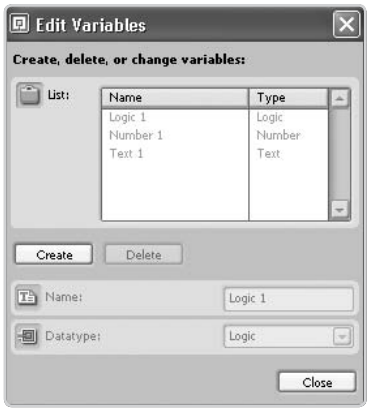


Figure 11-2: The Edit Variables dialog

Change the default name to something more meaningful; more descriptive variable names will make your program much easier to understand. For example, the RedOrBlue-Count program uses the variable names *Total Red* and *Total Blue*, which are good descriptions of how the variables are used.

You'll also need to select the data type for your new variable. The choices are Number, Text, and Logic, which are the same data types supported by data wires.

The Variable block will only let you store values of the selected data type in your variable. For example, if you try to connect a data wire with a text value to a Variable block set to use a number variable, the wire will appear broken. See "Understanding Data Types" on page 104 for more information on data types.

NOTE **To change a variable's data type after it has been created, select it in the list and then select the new data type. However, any data wires using that variable will break and will need to be fixed before your program will run.**

To change the name of an existing variable, select its name in the list, and enter a new name. Any Variable blocks that use the variable will be automatically changed to use the new name.

To delete a variable, select it in the list, and press the Delete button. If your variable is being used by one of the Variable blocks in your program, you won't be able to delete it, and you should see the dialog shown in Figure 11-3. To delete a variable, first remove any Variable blocks that use it.

Figure 11-3: Error deleting a variable

the variable block

The Variable block, in the Data group on the Complete Palette (Figure 11-4), stores and retrieves variable values. The Variable block should appear in your program as shown in Figure 11-5.

Figure 11-4: The Variable block on the Complete Palette

Figure 11-5: The Variable block

When working with a Variable block, you need to tell NXT-G which variable to use and whether to read to or write from the variable. To do so, use the Configuration Panel (Figure 11-6) to select the variable from the list, and then set the Action item to either Read or Write. When Read is selected, the variable's current value is put on the data wire attached the output data plug.

When Write is selected, a value is stored in the variable. That value can be supplied either using the Configuration Panel or via an input data wire. You'll usually use the Configuration Panel to set a variable's initial value at the beginning of a program and then use a data wire if you want to change the value later in the program.

You can tell which variable a Variable block is using, and whether it is being read or written, by the way the block is displayed in the Work Area. This feature makes it easy to tell what the block is doing without having to look at the

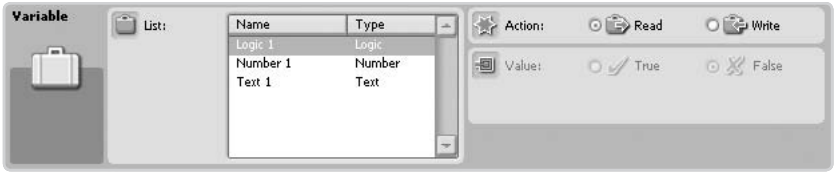


Figure 11-6: The Variable block's Configuration Panel

Figure 11-7: Variable name and the Action setting displayed on the block

VARIABLE NAMES

Choosing meaningful variable names is a simple way to make your program easier to understand. A name like *Total Red* makes it clear how the variable is being used. By the same token, avoid names that are too short, such as *TtlR* or just *R*. These names may make perfect sense to you while you're writing the program, but the meaning won't be obvious to someone else. Also, use a consistent naming scheme. For example, if you use *Total Red* for the number of red objects, then you should use *Total Blue* for the number of blue objects, not something a little different such as *Blue Count*.

The space available for the variable name on the Variable block and in the Configuration Panel is limited, so avoid using really long variable names. For example, if you used *Total Number of Red Objects* and *Total Number of Blue Objects* for variable names, then the Configuration Panel would look like Figure 11-8, making it difficult to pick the correct variable. In addition, Variable blocks using these two variables would be displayed as in Figure 11-9, and you can't tell which variable is being used by each block.

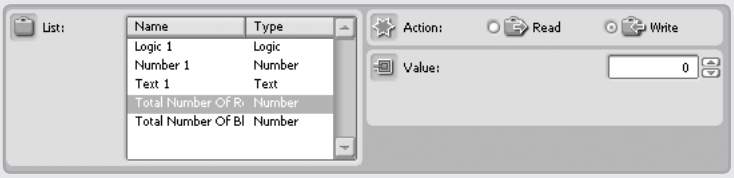


Figure 11-8: Configuration Panel with long variable names

Figure 11-9: Can't tell which variable is being used

Configuration Panel. For example, Figure 11-7 shows two Variable blocks that use the Total Red variable; the one on the left has Action set to Write, and the one on the right has Action set to Read.

the RedOrBlueCount program

In this section, I'll take you through the steps for creating the RedOrBlueCount program from the original RedOrBlue program. You'll create the two variables and then use them in the program to count the number of red and blue objects.

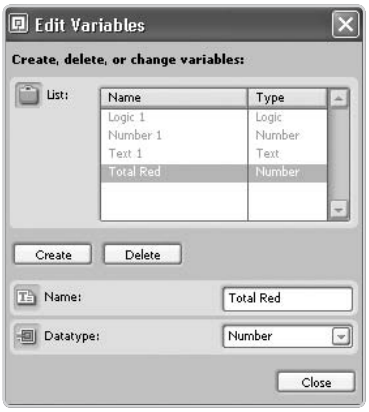
creating the variables

To create the two variables, follow these steps:

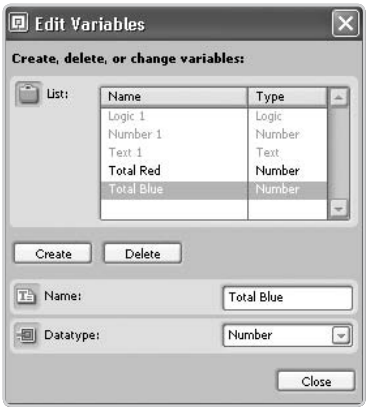
1. Open the RedOrBlue program.
2. Select **File ▸ Save As** from the menu to save the program as *RedOrBlueCount*.
3. Select **Edit ▸ Define Variables** from the menu to open the Edit Variables dialog. It should look like this:

4. Click the **Create** button.

5. Enter the name **Total Red**, and select **Number** for the data type. The dialog should look like this:



6. Click the **Create** button again.
7. Enter the name **Total Blue**, and select **Number** for the data type. The dialog should look like this:



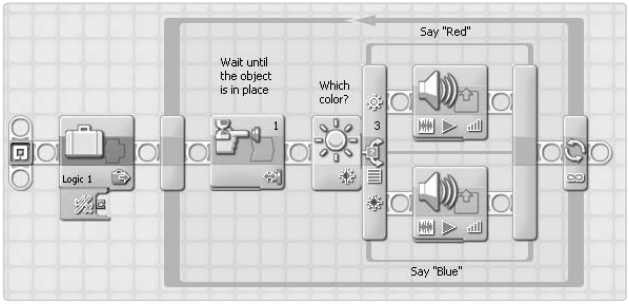
8. Click the **Close** button to close the dialog.

At this point, the two variables should have been created, and they are ready to be used in the program.

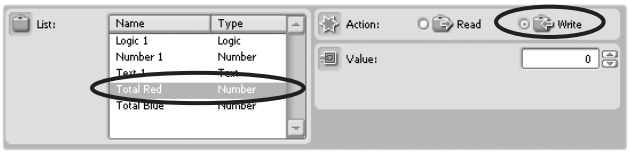
initializing the variables

To count the number of red objects, the value of the Total Red variable needs to start at zero and increase by one each time a red object is detected. When you create a Number variable, it will seem to have zero for a starting value; however, I always set a variable's initial value just to be sure. To initialize the variables to zero for the RedOrBlueCount program, you'll place two Variable blocks at the start of the program to write 0 to the Total Red and Total Blue variables, as follows:

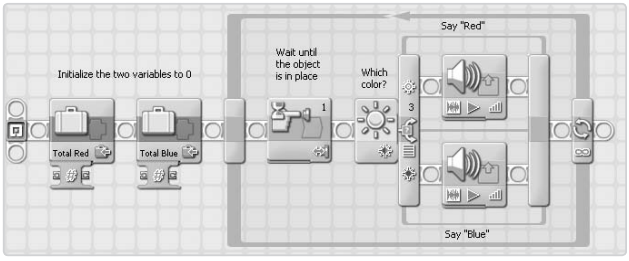
1. Place a Variable block at the beginning of the program, to the left of the Loop block. The program should look like this:



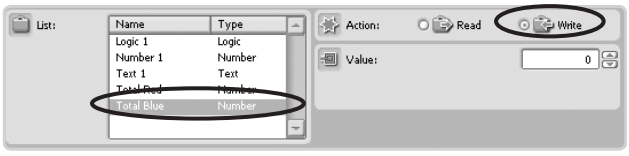
2. In the Variable block's Configuration Panel, select **Total Red** from the list, and set Action to **Write**. (The value should default to 0, so you don't need to change that.) The Configuration Panel should now look like this:



3. Add another Variable block before the Loop block.



4. Select **Total Blue** from the list of variables, and set Action to **Write**. The Configuration Panel for the new block should look like this:



initializing the display

As your program runs, it will display its count of the number of red and blue objects on the NXT's screen. When the program first starts, it should display `Red: 0` and `Blue: 0` and then update the display as the count increases. Listing 11-1 shows the pseudocode for the program, with the parts you've added to the RedOrBlue program in bold.

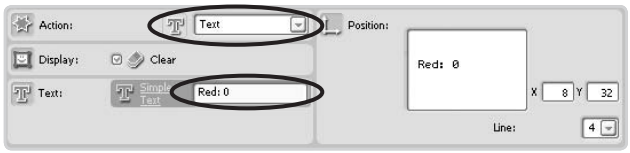
```
set Total Red to 0
set Total Blue to 0
display "Red: 0"
display "Blue: 0"
begin loop
   wait for the Touch Sensor to be bumped
   if the object is red then
      use a Sound block to say "Red"
      read the Total Red value
      add one to the Total Red value
      write the new value to Total Red
      display "Red: " followed by the Total Red
         value
   else
      use a Sound block to say "Blue"
      read the Total Blue value
      add one to the Total Blue value
      write the new value to Total Blue
      display "Blue: " followed by the Total Blue
         value
   end if
loop forever
```

Listing 11-1: The RedOrBlueCount Program

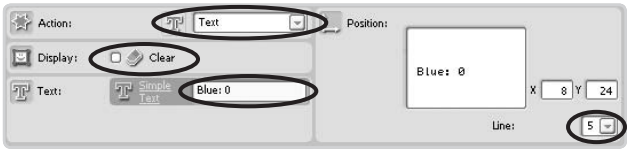
displaying the initial values

To display the initial values, use two Display blocks placed before the Loop block, like so:

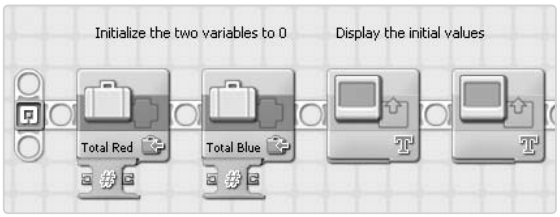
1. Add a Display block after the second Variable block. Set Action to **Text**, and set the text to **Red: 0**.



2. Add another Display block after the first one. Set Action to **Text**, and set the text to **Blue: 0**.
3. Uncheck the **Clear** option, and set Line to **5** to avoid erasing the text from the first Display block.



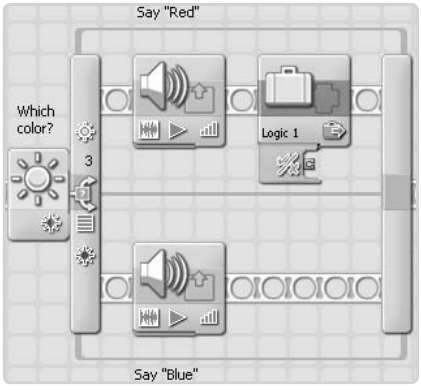
The beginning of the program should now look like this:



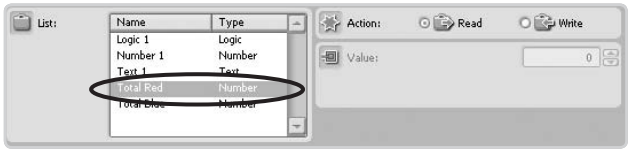
counting the red objects

Now comes the interesting part. When a red object is detected, you want the program to add one to the Total Red variable and display the new value. To make that happen, you'll use three blocks: a Variable block to put the current value on a data wire, a Math block to add one to the current value, and a second Variable block to store the new value. Here's how to do that:

1. Add a Variable block to the Switch block's upper Sequence Beam, after the Sound block. The Switch block should look like this:

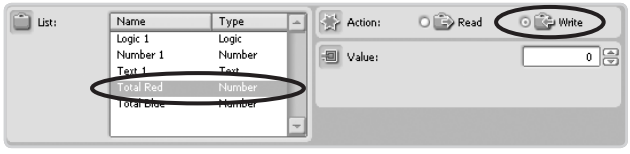


2. Select **Total Red** from the list of variables. (You don't need to change the Action setting, because it should already be set to **Read**.)

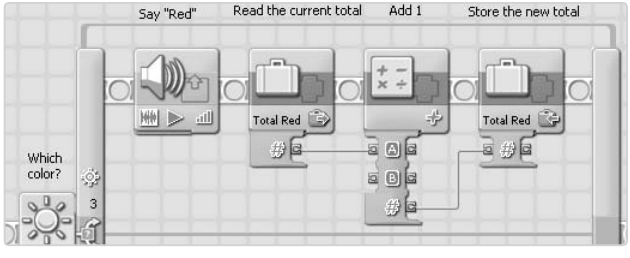


3. Add a Math block after the Variable block, and set the B value to **1**.

4. Add another Variable block after the Math block. Select **Total Red** from the list of variables, and set Action to **Write**. You don't need to set the value in the Configuration Panel because it'll be set using a data wire from the Math block.

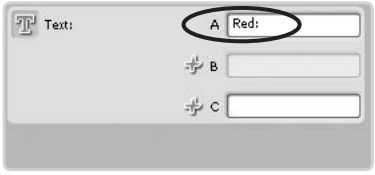


5. Connect the blocks by drawing a data wire from the output plug of the first Variable block to the Math block's A input plug.
6. Draw a data wire from the output plug of the Math block to the input plug of the second Variable block. This section of the program should now look like this:

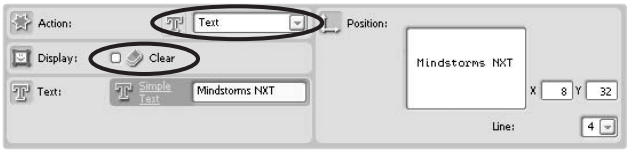


Once the total has been updated, you want the program to display the new value. To do so, you'll use a Number to Text block to convert the total to a text value, a Text block to add the label, and a Display block to write the labeled value to the display. (This is the same technique that you used in the SoundMachine program.)

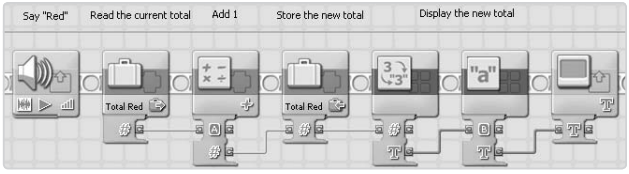
7. Add a Number to Text block after the second Variable block.
8. Connect the output plug of the second Variable block to the Number to Text block's input plug.
9. Add a Text block after the Number to Text block, and set the A value to **Red:** . (Be sure to include a space after the colon so that the number won't be squished up against the label.)



10. Connect the Number to Text block's output data plug to the Text block's B input plug.
11. Add a Display block after the Text block, and set Action to **Text**. Be sure to uncheck the **Clear** option so that you won't erase the blue total.



The upper Sequence Beam of the Switch block should now look like this:



The section of code that you've just created contains two patterns that you'll see often. To change a variable's value, a Variable block is used to read a value, one or more blocks are used to modify that value, and a second Variable block is used to write the new value to the variable. The second pattern is used to display a number using a Number to Text block followed by a Text block to add a label followed by a Display block.

NOTE **Before continuing, test your program to make sure it can correctly count and display the total number of red objects. If you made a mistake writing this code, there's a good chance that you will make the same mistake in the code for counting blue objects, in which case you'd need to fix two bugs. Testing your program as you add each piece makes finding bugs easier and reduces the chance of repeating the same bug.**

counting the blue objects

The code for counting the blue objects is almost identical to that used for the red objects, with these differences:

* The Variable block needs to use Total Blue instead of Total Red.
* The label in the Text block should be *Blue:* (with a space after it).
* The Display block needs to use line 5.

Instead of writing the code by hand, you can copy the code for counting the red objects and then make a few changes. Here's an easy way to duplicate the code from the upper Sequence Beam to the lower Sequence Beam:

1. Select the six blocks on the upper Sequence Beam (from the first Variable block to the Display block) by drawing a selection rectangle around them or by selecting the Variable block and then holding down the SHIFT key while clicking the other blocks.
2. While holding down the CTRL key, click one of the blocks, and drag it to the lower Sequence Beam. (When you drag with the CTRL key down, the blocks are copied instead of moved.)

Once the blocks have been copied and are in place, you can make the necessary changes to the Configuration Panel settings as listed earlier. Figure 11-10 shows the section of the program on the lower Sequence Beam, and Figures 11-11 through 11-14 show the Configuration Panels for the Variable, Text, and Display blocks. (You don't need to change the settings for the Math and Number to Text blocks.)

With these blocks copied and the settings changed, your program should now correctly count and display the totals for both red and blue objects.

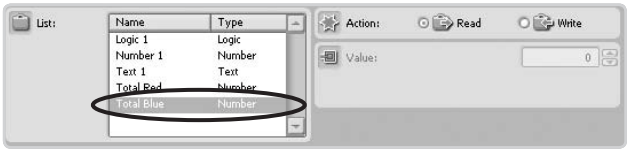


Figure 11-11: Configuration Panel for the first Variable block

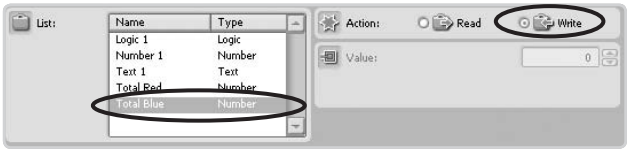


Figure 11-12: Configuration Panel for the second Variable block

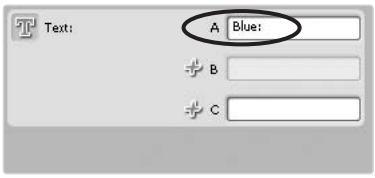


Figure 11-13: Configuration Panel for the Text block

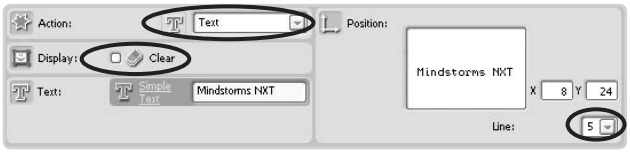


Figure 11-14: Configuration Panel for the Display block

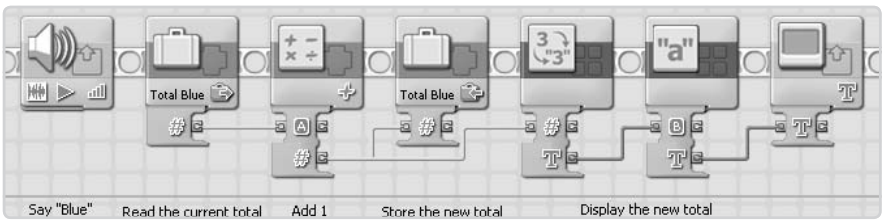


Figure 11-10: Code for counting blue objects

grouping common settings

It's not unusual for a program to use several blocks with the same settings. For example, the WallFollower program contains seven Move blocks that all use the same Power setting. If you decide to change the setting from 35 to 45, you need to make sure you change that setting on all seven blocks. That can become problematic when you're trying several different Power settings, in search of the perfect balance between speed and accuracy; it's almost inevitable that you'll forget to change the setting on one of the blocks.

To avoid this problem, use variables to group common settings. For example, Figure 11-15 shows a modified version of the AroundTheBlock program from Chapter 4. Here you use a variable (named *Power*) to control both Move blocks, rather than set the Power value for each Move block separately using the Configuration Panels. This way, when testing different Power settings, you would need to change only the first Variable block, instead of changing each Move block. In addition, it prevents you from changing a setting on one block and not the other. These benefits become even more apparent when you're working with larger programs, such as WallFollower where you can change one block instead of seven.

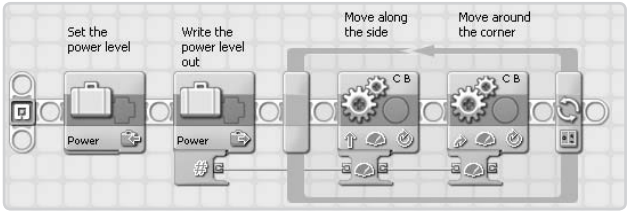


Figure 11-15: Using a variable to set the power level

NOTE **This idea of changing the settings in only one place is called the Don't Repeat Yourself (DRY) principle. It's a common theme in computer programming that helps reduce errors by reducing the number of places you can make a mistake.**

replacing long data wires with variables

Variables can replace really long data wires in your programs. Instead of using a long data wire, you can use a pair of Variable blocks and a variable with a meaningful name. You use the first Variable block to write the value to a variable just after the block that creates the value. Later in the program, you can read the variable using a second Variable block placed next to the blocks that use the value.

When you can see both ends of a data wire, it's usually easy to understand how it's being used. When a program becomes too long to fit on the screen or you can't see where a data wire starts, it's much more difficult to follow the logic of the program. A short data wire attached to a Variable block (which displays the name of the value) can make a program's logic more apparent.

the LightPointer program

The LightPointer program presented next demonstrates how to use variables to remember values that you want to use later in your program. This program uses the Color or Light Sensor to point the TriBot at a light source by spinning the robot in a circle and remembering where the sensor detected the brightest light. The code you develop here could be used as part of a larger program, such as a fire-fighting simulation or a robot that tracks the position of the sun to adjust a solar panel.

In the following text and images, I'll use the Color Sensor (in Light Sensor mode) to read the level of ambient light. The program also works just as well with the Light Sensor. The sensor can be placed either at the front of the TriBot (Figure 11-16) or on its side (Figure 11-17). You can, of course, adjust the placement of the sensor depending on the light source.

Figure 11-16: Color Sensor on the front of the TriBot

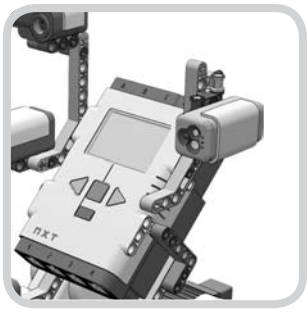


Figure 11-17: Color Sensor on the side of the TriBot

The program does two things: It searches for a light source, and then it points the TriBot at that light source. For the first part, the TriBot slowly spins in a circle as its sensor constantly measures the amount of ambient light. Each reading is compared with the largest reading seen so far, and when a larger reading is measured, the position is recorded as that of the light source.

Figure 11-18 shows how the sensor reading changes as the TriBot spins. The robot starts facing away from the light, so the reading is low (10). As the robot spins toward the flashlight, the sensor reading increases to 40, as shown in the second image. When the TriBot is pointing directly at the flashlight, the reading will be at its highest level (70 in this example). The sensor reading will then decrease as the TriBot spins past the flashlight, as shown in the final image.

The second part of the program turns the TriBot back to the position with the largest reading, which should result in the robot pointing at the light source.

defining the variables

This program needs two variables, both of which will hold numerical values. The first one, MaxReading, tracks the brightest sensor reading seen so far. The other, Position, holds the robot's position when the reading was taken. Figure 11-19 shows the Edit Variables dialog with these variables defined.

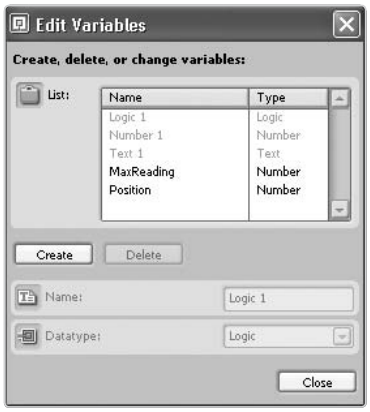


Figure 11-19: Defining the MaxReading and Position variables

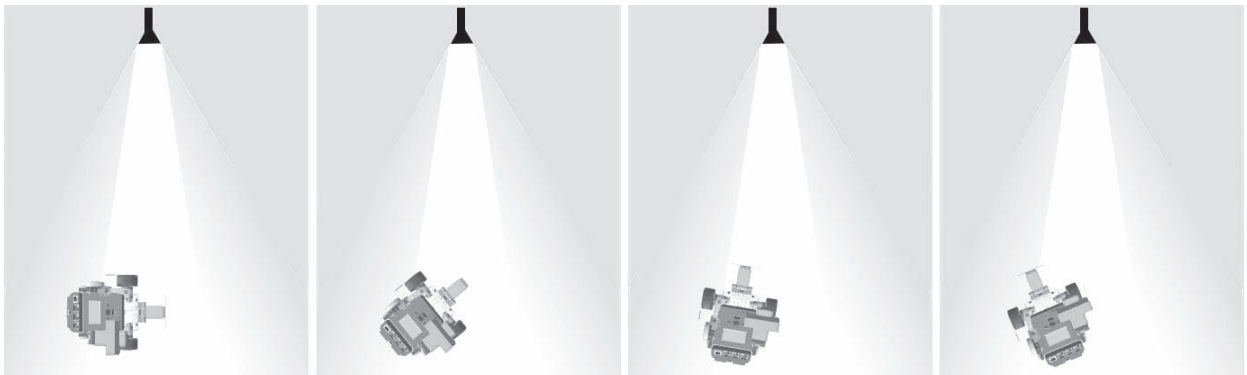


Figure 11-18: Sensor readings at four positions

finding the light source

The first step in finding the light source is to spin the robot around. A Move block with the Steering slider set all the way to one side will accomplish this nicely: Put the slider all the way to the left (toward the C motor); the rotation values for the B motor increase as the TriBot moves, and those for the C motor decrease. I find it easier to work with positive numbers, so I'll use the rotation values from the B motor to track the robot's position.

The TriBot should spin in a complete circle so that it can find the light source in any direction. A little experimentation shows that a Duration setting of 1100 degrees moves the robot just past a full circle, although this value doesn't have to be exact.

Because the program needs to read the sensor while the robot is spinning, you can't just use a Move block with Duration set to 1100 degrees. Instead, you need to use a Move block with Duration set to Unlimited followed by a Loop block that stops when the reading from the B motor's Rotation Sensor reaches 1100.

NOTE **For the balloon tires you can use 800 degrees for a full circle.**

As the TriBot spins, the program will compare the reading from the Color Sensor with the highest reading seen so far. If a higher reading is found, MaxReading is set to the new (higher) value, and Position is set to the B motor's position. Listing 11-2 shows the pseudocode for this section of the program. Figure 11-20 shows the variable values at the positions shown in Figure 11-18.

```
start the robot spinning slowly
begin loop
   if Color Sensor reading > MaxReading then
      MaxReading = Color Sensor reading
      Position = B motor Rotation Sensor reading
   end if
loop until B motor Rotation Sensor > 1100
stop the motors
```

Listing 11-2: Finding the Light Source

initializing the values

Before writing the code for the first part of this program, think about the values the variables should have at the start of the program. Even though the code used to initialize the values comes first in the program, you'll usually need to first design the program (or at least its major parts) in order to determine what needs to be initialized.

The MaxReading variable holds the highest reading from the Color Sensor, in the range of 0 to 100. Setting MaxReading to 0 at the beginning of the program ensures that the sensor reading and the robot's position are set the first time the sensor reads a value greater than 0.

Even though the Position variable will be set when the sensor first detects light, it's still a good idea to give it an initial value. (Initializing all your variables is a good programming practice that can help avoid some bugs that can be difficult to find.) In the code shown in the following section, the Position variable is set to 0 at the start of the program.

In addition to the two variables, the code in Listing 11-2 uses the Rotation Sensor for motor B. To make sure that

Figure 11-20: MaxReading and Position variable values as the robot spins

the loop works correctly, reset the Rotation Sensor at the beginning of the program. Although the Rotation Sensor will automatically be set to zero when the program starts, to reuse the code for this program in another program, you will need to reset the Rotation Sensor before the Move block starts spinning the robot. Your programs will be much easier to reuse if you explicitly initialize things instead of relying on automatic initialization when the program starts.

the LightPointer program, part 1

As discussed in "Initializing the Values" on page 140, when the LightPointer program starts, it initializes the two variables and resets the B motor's Rotation Sensor, as shown in Figure 11-21. Figures 11-22 through 11-24 show the Configuration Panels for these blocks.

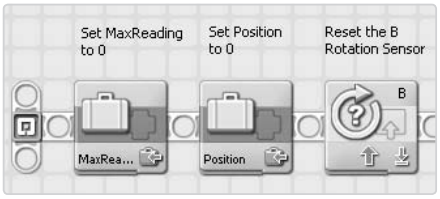


Figure 11-21: Setting the initial values

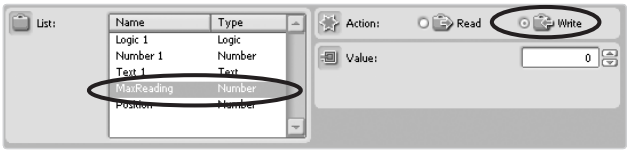


Figure 11-22: MaxReading set to 0

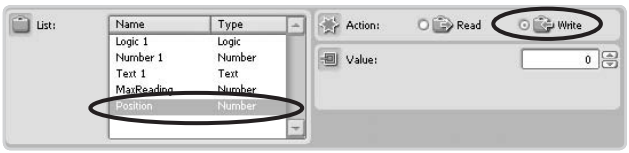


Figure 11-23: Position set to 0

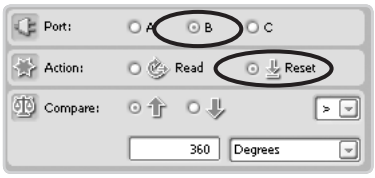


Figure 11-24: Resetting the B motor's Rotation Sensor

finding the light source

With initialization complete, you can begin writing the code for locating the light source as shown in the pseudocode developed in "Finding the Light Source" on page 140. First, start the TriBot spinning by using a Move block, as shown in Figure 11-25. Set Power to 20 to make the robot spin slowly enough so that it doesn't miss the light source. Figure 11-26 shows the Configuration Panel for the Move block.

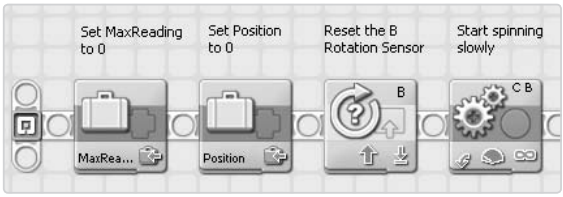


Figure 11-25: Starting the robot spinning

Figure 11-26: Configuration Panel for the Move block

The next part of the program, shown in Figure 11-27, is much more interesting. In this section, the Loop block keeps the robot spinning until the Rotation Sensor reads greater than 1100 degrees, using the settings shown in Figure 11-28. Each time through the loop, the Color Sensor's reading is compared with the MaxReading value. If the sensor reading is greater than the current MaxReading value, the code in the Switch block is executed. This code updates

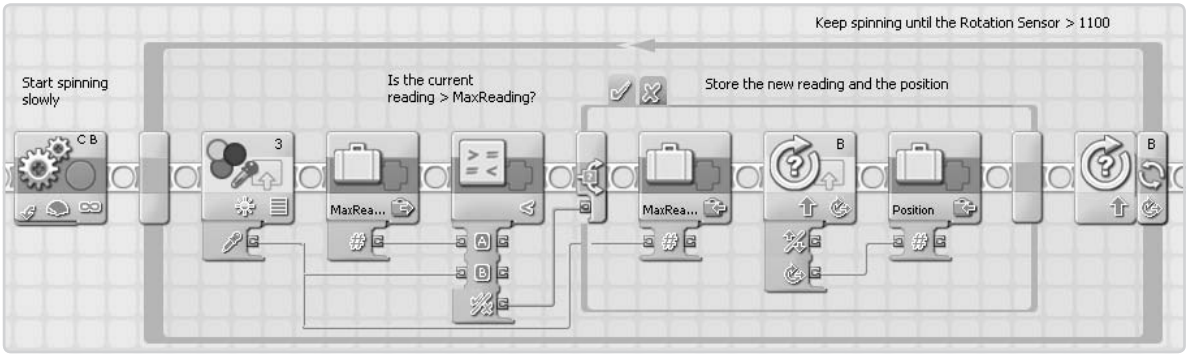


Figure 11-27: Finding the brightest source of light

the MaxReading variable and sets the Position variable to the current motor position using the Rotation Sensor. There are no blocks on the other tab of the Switch block.

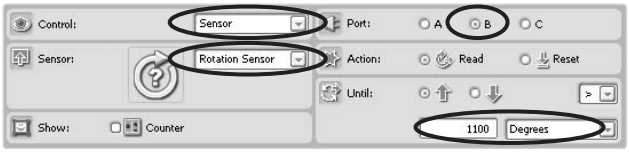


Figure 11-28: Looping until the Rotation Sensor reads greater than 1100

To use the Color Sensor to measure the light, set it to use Light Sensor mode, and turn the light off, as shown in the Configuration Panel in Figure 11-29.

NOTE **Unfortunately, the Color Sensor block's output plug is labeled *Detected Color*, which doesn't really make sense when using Light Sensor mode. The help file adds to the confusion a bit by neglecting to mention that this plug is used for the intensity when using Light Sensor mode.**

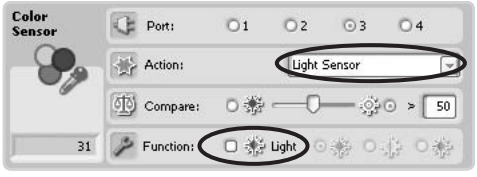


Figure 11-29: Reading the ambient light level using the Color Sensor

If you're using the Light Sensor instead of the Color Sensor, configure the Light Sensor block, as shown in Figure 11-30.

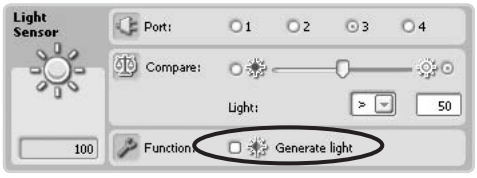


Figure 11-30: Reading the ambient light level using the Light Sensor

Figure 11-31 shows the Configuration Panel for the Variable block that reads the current MaxReading value so that it can be compared with the sensor reading.

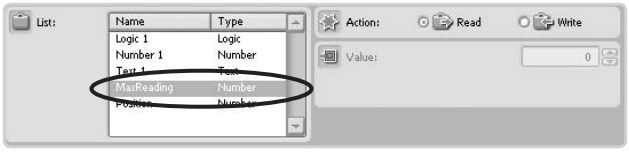


Figure 11-31: Reading the MaxReading variable

The Comparison block (shown in Figure 11-32) compares the MaxReading value, connected to the A input plug, with the reading from the Color (or Light) Sensor, connected to the B input plug. The Switch Block (shown in Figure 11-33) uses the result of this comparison to decide whether a new maximum value has been seen. (The Switch block uses Tabbed View so that the data wire from the Color Sensor block can be used by the Variable block inside the Switch block.)

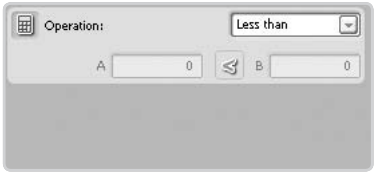


Figure 11-32: Is MaxReading less than the reading from the sensor?

Figure 11-33: The Switch block Configuration Panel

If the MaxReading value is less than the sensor reading, then the three blocks within the Switch block will be executed. First the Variable block (shown in Figure 11-34) writes the sensor reading to the MaxReading variable. Then the Rotation Sensor block (shown in Figure 11-35) reads the current motor position, and the Variable block (shown in Figure 11-36) saves this value in the Position variable.

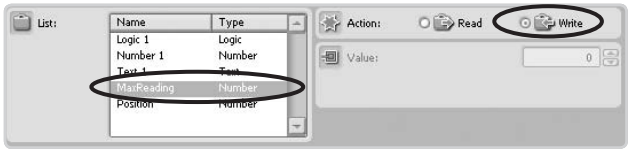


Figure 11-34: Saving the new maximum light level

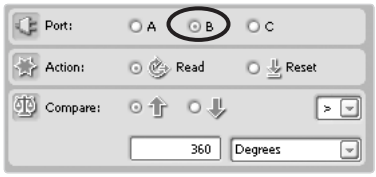


Figure 11-35: Reading the position of motor B

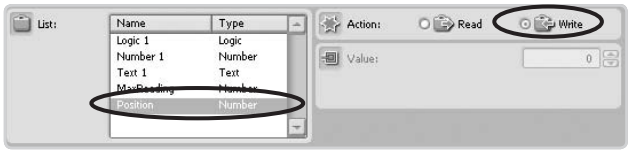


Figure 11-36: Saving the new position

stopping the motors

Once the TriBot has spun all the way around, stop the motors by using a Move block with the direction set to Stop. Figures 11-37 and 11-38 show the placement of the Move block and its Configuration Panel.

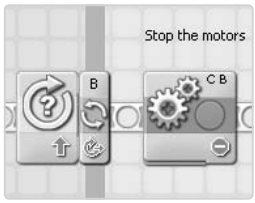


Figure 11-37: The Move block following the Loop block

Figure 11-38: Stopping the motors

the LightPointer program, part 2

Once the first part of the program has completed, the Position variable should contain the position of motor B where the brightest light was detected. The second part of the program should make the TriBot spin in the opposite direction to return to that position.

A Move block spinning the TriBot backward will cause the Rotation Sensor value for the B motor to decrease. To make the TriBot stop at the correct place, it needs to keep spinning as long as the Rotation Sensor reading is greater than the Position value. Here is the pseudocode for this section of the program:

```
start the robot spinning slowly in the opposite
   direction from the first move
read the Position variable
begin loop
   read the Rotation Sensor for motor B
loop until the Rotation Sensor reading reaches the
   Position value
stop the motors
```

Figure 11-39 shows this section of the program, and Figures 11-40 through 11-44 show the Configuration Panels for the blocks. Notice that you can use the Rotation Sensor block to both read the value and perform the comparison by using a data wire to set the value stored in the Position variable as the Rotation Sensor block's target value. The comparison is set to >, so the output of the Yes/No data plug will be true until the robot reaches the target position. When

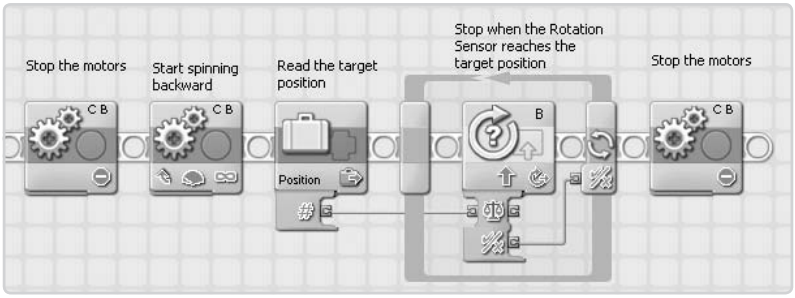


Figure 11-39: Spinning back to the saved position

the output becomes false, the Loop block will complete, and the Move block will stop the motors.

Figure 11-40: Starting spinning backward

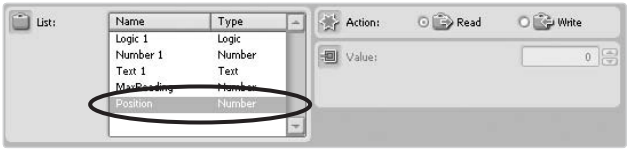


Figure 11-41: Reading the target value from the Position variable

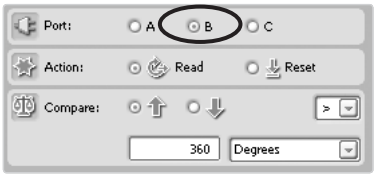


Figure 11-42: Comparing the Rotation Sensor with the target

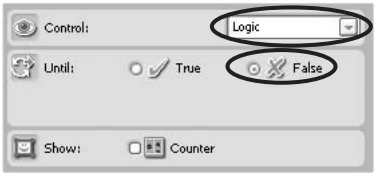


Figure 11-43: Stopping the loop when the target is reached

Figure 11-44: Stopping the motors

When this program runs, the TriBot should slowly spin all the way around and then reverse direction and stop with the robot pointing at the brightest light source. Try testing this behavior in a dark room with a flashlight shining at the robot.

constants

As you write more programs, you may find that you use some values repeatedly. A value that doesn't change is called a *constant*. For example, the number of degrees needed to spin the TriBot in a full circle is a constant. The value doesn't change, as long as you don't change the way the robot is built. You can use this value in any program that spins the robot all the way around, and by adding a Math block, you can easily spin the TriBot halfway around, as in the ThereAndBack program from Chapter 4; or you can have the TriBot make a quarter turn, as in the WallFollower program from Chapter 7. Some other constants you might find useful are the distance the robot moves in one rotation of the motor or the mathematical constant pi.

To make working with these repeated values easier, NXT-G 2.0 adds support for constants. NXT-G constants are similar to variables, except that a constant's value doesn't change. Using constants, you can define a value once and then use it in many programs.

managing constants

Use the Edit Constants dialog (shown in Figure 11-45) to create, edit, or delete constants. To open the dialog, select Edit ▸ Define Constants from the menu.

Figure 11-45: The Edit Constants dialog

The Edit Constants dialog is similar to the Edit Variables dialog except in these ways:

- When you create a constant, you give it a value as well as a name and data type.
- To change an existing constant, you select it in the list and click the Edit button. When you're finished making changes to the name, value, or data type, you either click OK to save the changes or click Cancel to discard them. (The Edit Variables dialog doesn't have an Edit button, and any changes you make are immediately applied.)
- Changing the name of a constant won't change the name in any Constant blocks that use the constant.
- You can delete a constant even if it is being used in a program.

NOTE **If you change a constant's value, be sure to save any open programs that use that constant. Until you save a program, it will continue to use the old value.**

Unlike variables, which are available only to the program that defines them, once you define an NXT-G constant, you can use it in any program. Each NXT-G program has its own list of variables, but all share the same list of constants.

the constant block

To access constants in your program, use the Constant block (Figure 11-46) at the end of the Data group on the Complete Palette. The Constant block will be displayed in your program, as shown in Figure 11-47. This block looks like the Variable block, with a lock added to show that the value can't be changed.

Figure 11-46: The Constant block on the Complete Palette

Figure 11-47: The Constant block

When you first add a Constant block to your program, the Configuration Panel looks like Figure 11-48. The Action item determines how you will define the value to use, either selecting a constant from the list created using the Edit Constants dialog or creating a custom constant.

choose from list

Although the Custom option is the default, I'll cover the Choose from list option first because it's simpler to use, and I recommend it as the one to use in your programs.

When you select Choose from list, the Configuration Panel lists the constants that have been created using the Edit Constants dialog (as shown in Figure 11-49). Simply select the constant you want to use from the list; unlike with variables, there is no Read or Write option.

The Constant block always reads the constant value and puts it on the data wire attached to the output data plug. The name of the constant you select will be shown on the

Figure 11-48: The Constant block's Configuration Panel

block, and the data type of the output plug will change to match the type of the constant, as shown in Figure 11-50.

Figure 11-49: Choosing a constant from the list

Figure 11-50: The Constant block after selecting Degrees To Spin

To use the constant value, simply connect a data wire to the Constant block's output data plug, just as you would when using a Variable block to read a variable.

custom

When the Action is set to Custom, the settings in the Configuration Panel define a new constant, specifying the data type, value, and name (see Figure 11-48). For reasons I'll explain, when writing your own programs, you should use the Choose from list option. However, it's important to know how the Custom option works because there are two circumstances where NXT-G will automatically change the Action setting for a Constant block from Choose from list to Custom:

1. When you delete a constant using the Edit Constants dialog, any Constant block that uses that constant will change: The Action will be set to Custom, and the name, data type, and value will be set to match the way the constant was defined the last time the program was saved.
2. A similar situation can occur if you open a program created on a different computer. For example, say your friend Peter writes a great program that use constants and gives you a copy to use with your robot. When you open the program on your computer, you may encounter problems if Peter used constants that you haven't created on your computer. Any Constant blocks that refer to constants that aren't defined on your computer will have be changed as described earlier.

Although in either case the changes to the Constant block settings won't affect its behavior, as I explain next, you will need to be careful if you want to modify Peter's program.

working with custom constants

Constants are meant to be defined with the Edit Constants dialog, but custom constants are necessary when you're faced with either of the two cases described earlier, where the constants used by a program are not in the list of defined constants. When working with a program that uses custom constants, be aware of the following behaviors.

- Changing the value of a custom constant won't change the value in other Constant blocks that use the same name. For example, if Peter's program uses a constant named *RoomSize* in three places and you want to change the value, make sure to change the value in all the Constant blocks that use RoomSize. Changing the value in one Constant block won't affect the others, and you'll end up using different values, which is exactly what you don't want from a constant.
- Creating a constant using the Edit Constant dialog will automatically change any Constant block that uses the same name and data type to use Choose from list. The value used by the program will be the one set in the Edit Constant dialog, not the one set in the Configuration Panel when Custom was selected.

Although you can use the Constant block to create custom constants, I don't recommend doing so because you will run into the issues mentioned earlier if you are not very careful about the name you use. Similar problems may occur if you write a program using a custom constant and give it to Peter, if he happens to have a constant defined that uses the same name. The best approach is to use the Edit Constants dialog to define the constants when writing your programs and when using a program written on another computer.

conclusion

NXT-G variables let you store the data that your program uses. Use the Edit Variables dialog to create variables, and use the Variable block to access the variables within your program. The programs presented in this chapter have shown you a few ways to use variables in your programs. Like data wires, variables add a lot of flexibility to the NXT-G language and are essential for solving many types of problems. You'll see variables used in many of the programs in the coming chapters.

NXT-G 2.0 allows you to use constants in addition to variables. Constants make it more convenient to use values that don't change, especially if you use them in many different programs.

12

the NXT buttons and the display block

The NXT has three buttons and a small display screen that you can use to interact with your programs, similar to the way you use a keyboard and monitor with most computers. In this chapter, you'll learn how your programs can use the buttons on the NXT. You'll also learn about some new features of the Display block that will give you more control over the NXT's screen.

The chapter begins with two small programs that show you how to use the buttons to adjust the value of a variable. The first program displays the value using text, as you've done in previous programs. The second program shows you how to display the same information using images. The chapter then shows a final program, which uses a Display block and the TriBot's two wheels to build a sketch pad.

the NXT buttons

You can use the three large buttons on the front of the NXT (shown in Figure 12-1) to control your program. For example, you can make the program wait for you to press a button or continue working until you press a button. You can also write a program that lets you use the buttons to choose different program options. The buttons work like the Touch Sensor, and your program can detect whether a button is pressed, is released, or has been bumped (pressed and quickly released). The other button (the Exit button) can't be used by a program; pressing it while your program is running will end the program.

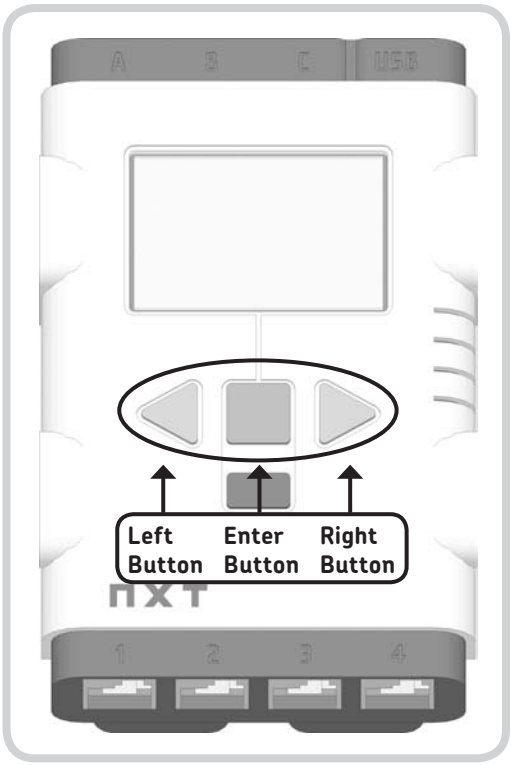


Figure 12-1: The NXT buttons

To use a button with the Wait, Switch, or Loop blocks, choose NXT Buttons from the list of sensors (shown in Figure 12-2). You can also work with the buttons using the NXT Button block, which is similar to the other sensor blocks.

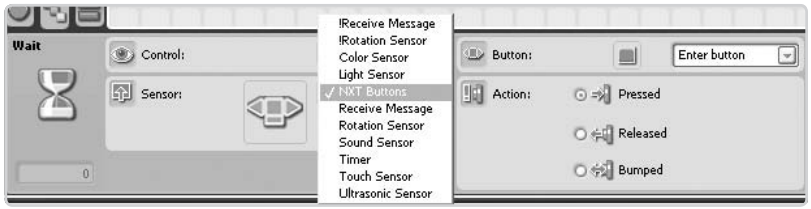


Figure 12-2: NXT Buttons in the list of sensors for the Wait block

the NXT button block

You'll find the NXT Button block on the Complete Palette with the other sensors, as shown in Figure 12-3. Figure 12-4 shows how the block should appear in your program.

Figure 12-3: The NXT Button block on the Complete Palette

Figure 12-4: The NXT Button block

Use the Configuration Panel (shown in Figure 12-5) to select the button you're interested in and the action you want to detect (Pressed, Released, or Bumped).

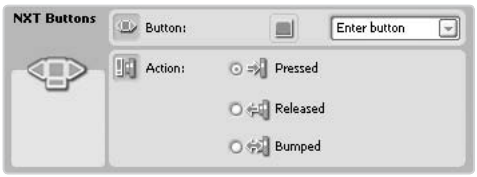


Figure 12-5: The NXT Button block's Configuration Panel

The Configuration Panels list the buttons as Enter button, Left button, and Right button. The Enter button is the orange square one in the middle, the Left button is the one on your left as you look at the NXT, and the Right button is the one on your right, as shown earlier in Figure 12-1.

the PowerSetting program

In Chapter 11, I mentioned that you could improve the WallFollower program by using a variable to store the Power setting used by the Move blocks. This would make changing the value easier because you would need to change it in only one place, instead of changing all seven Move blocks. What if you want to try several different values? This can still be a time-consuming process since you would need to change the Variable block and download the program for each value you want to test.

You can make testing different Power settings more convenient by using the buttons to select the Power setting before starting the main part of the program. In this section, I'll describe the PowerSetting program to show you how to do this. With just a few simple changes, you can use this code any time you want to change a value while the program is running.

The PowerSetting program uses a variable, called *Power*, to store the current value. This value is displayed on the screen, and the buttons are used to change the value. Pressing the Right button will increase the value, and pressing the Left button will decrease the value. Pressing the Enter button will accept the current value. Listing 12-1 shows the pseudocode for the program.

```
set Power to 25
begin loop
   display the current value
   if the Right button is bumped then
      Power = Power + 1
   end if
   if the Left button is bumped then
      Power = Power - 1
   end if
loop until the Enter button is bumped
```

Listing 12-1: The PowerSetting Program

Listing 12-1 describes the PowerSetting program at a higher level than the earlier listings. For example, `display the current value` will require four blocks: a Variable block to read the value, a Number to Text block to convert the value to text, a Text block to add a label, and finally a Display block to display the labeled value. (I've used this pattern for displaying values several times, so I haven't included the details in the listing.) Likewise, the line `Power = Power + 1` is a short way of saying, "Take the current value of the Power variable, add one to it, and store the result in the Power variable." In the PowerSetting program, you'll use the familiar three block groups to accomplish this, but the pseudocode is easier to understand using this more concise description.

defining the variable

The first step in building this program is to define one variable, Power, as shown in Figure 12-6.

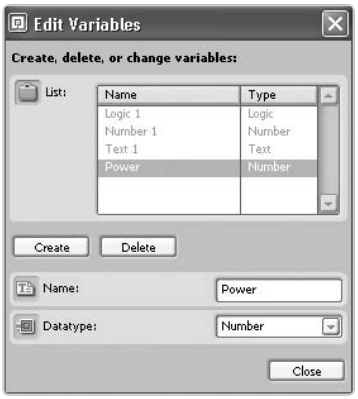


Figure 12-6: Defining the Power variable

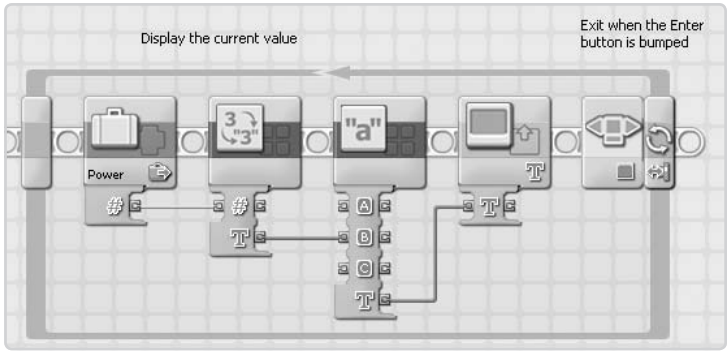


Figure 12-10: Blocks to display the current value

the initial value and the loop

Before using the Power variable, use a Variable block to set its initial value. For this example, I'll use 25, since that's the value used by the WallFollower program. Figure 12-7 shows the Variable block followed by the Loop block that will hold the rest of the program. The Loop block keeps repeating until the Enter button is bumped. Figures 12-8 and 12-9 show the Configuration Panels for these two blocks.

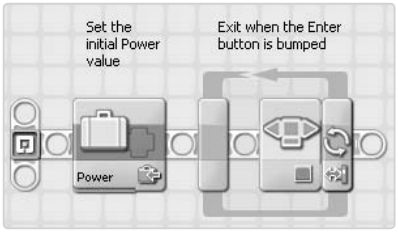


Figure 12-7: Initializing the Power variable and starting the loop

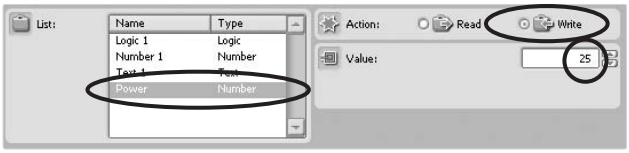


Figure 12-8: Setting Power to **25**

Figure 12-9: Looping until the Enter button is bumped

displaying the current value

Each time the loop repeats, the program displays the current value. Use a Number to Text block followed by a Text block to add a label to the value so that it looks like this: `Power: 25`. Figure 12-10 shows the code used to display the value.

When entering the label in the Text block, remember to put a space after the colon so that the number doesn't squish up against the label; the display should show `Power: 25` and not `Power:25`. Figures 12-11 through 12-13 show the Configuration Panels for the Variable, Text, and Display blocks. As usual, the Number to Text block uses all the default settings.

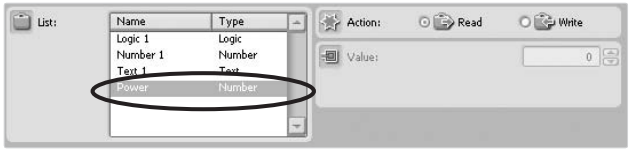


Figure 12-11: Reading the current Power value

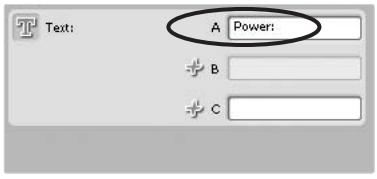


Figure 12-12: Adding a label to the value

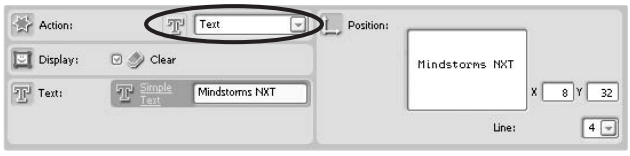


Figure 12-13: Displaying the labeled value

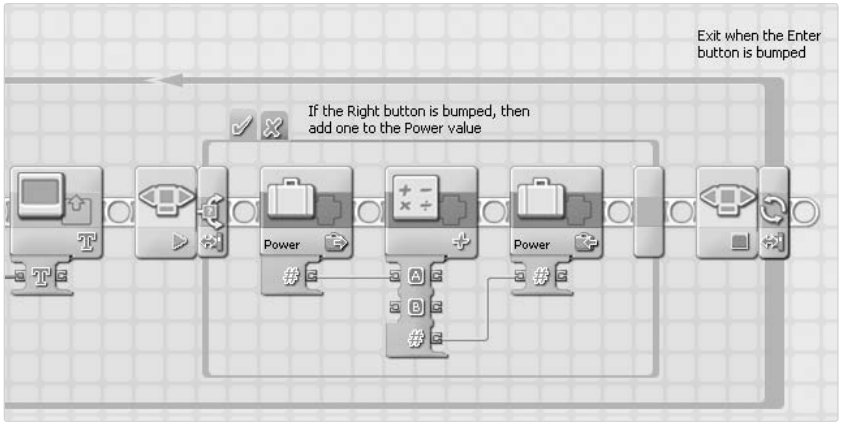


Figure 12-14: Adding one to Power if the Right button has been bumped

adjusting the power value

Once the current Power value has been displayed, you can use the NXT's Left and Right buttons to adjust it. Deal with the Right button first, using the code shown in Figure 12-14. The Switch block can tell whether the button has been bumped; if so, one is added to the Power variable.

The Switch block is configured to trigger when the Right button is bumped (as shown in Figure 12-15). The Flat view option is unchecked to make the program take up less space.

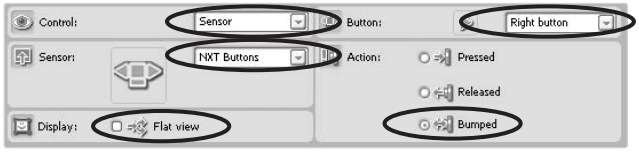


Figure 12-15: Has the Right button been bumped?

Figures 12-16 through 12-18 show the Configuration Panels for the Variable and Math blocks, which use the familiar pattern to add one to the Power variable. These blocks are on the Switch block's true tab because you want to run them only if the button has been bumped. There are no blocks on the false tab because you don't need the program to do anything when the button hasn't been bumped.

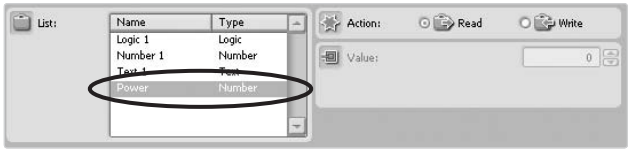


Figure 12-16: Reading the current Power value

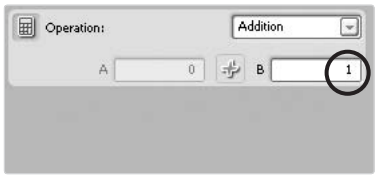


Figure 12-17: Adding one to the Power value

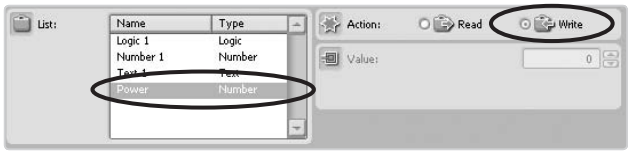


Figure 12-18: Storing the new Power value

The code for dealing with the Left button is almost identical, except that when this button is pushed, you need to subtract one from the Power value. The new code (shown in Figure 12-19) uses the same four blocks you used with the Right button with only two differences on the Configuration Panels: The Switch block uses the Left button, and the Math block operation is Subtraction. Figures 12-20 and 12-21 show the Configuration Panels for the Switch and Math blocks. The settings for the two Variable blocks are the same as those shown earlier in Figures 12-16 and 12-18.

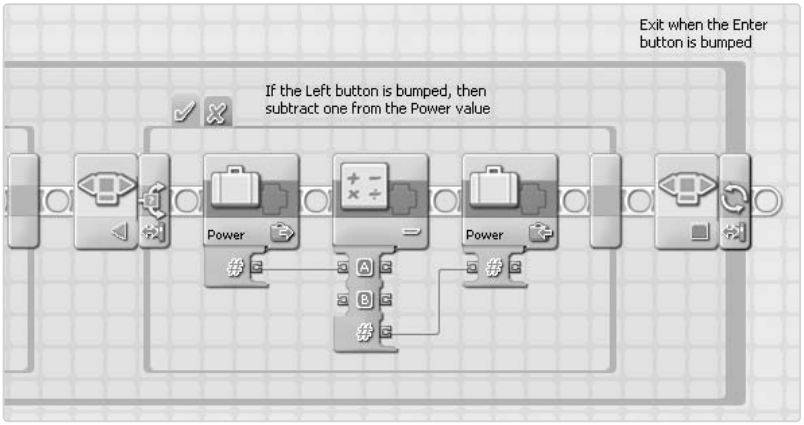


Figure 12-19: Subtracting one from Power if the Left button has been bumped

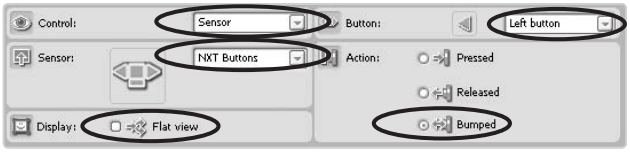


Figure 12-20: Has the Left button been bumped?

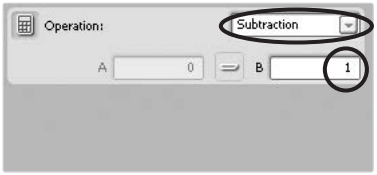


Figure 12-21: Subtracting one

testing the program

When you run this program, it should first display `Power: 25`. Press the Right and Left buttons to change this value. Press the Enter button, and the program should end.

When the program ends, the Power variable will be set to the value you've selected. In practice, you would use the blocks that make up this program at the beginning of a larger program to set a variable. Pressing the Enter button would start the main part of the program (instead of exiting).

making the program faster

The PowerSetting program works great when changing a value from 25 to 30, but what if you were to change it to a value like 80? Pressing the Right button 55 times can get a little annoying. Making a large change to the value is rather inconvenient because each time you press and release the button, the value changes only by one. How can you speed this up?

You need to press and release the button to change the value because the Action setting of the two Switch blocks is set to Bumped. The program might respond quicker if you just check for the button being pressed, instead of waiting for it to be pressed and released. What happens if you change the Action setting of both Switch blocks to Pressed, as shown in Figures 12-22 and 12-23?

Figure 12-22: Is the Right button pressed?

Figure 12-23: Is the Left button pressed?

The value certainly changes faster, but now it changes too fast. To make the program usable, you need to slow the loop down a little by adding a Wait Time block (Figures 12-24 and 12-25) at the end of the loop body. I've found that a setting of 0.2 seconds gives me a good balance between changing the value quickly and being able to stop at the value I want. Experiment with different values to see what works best for you.

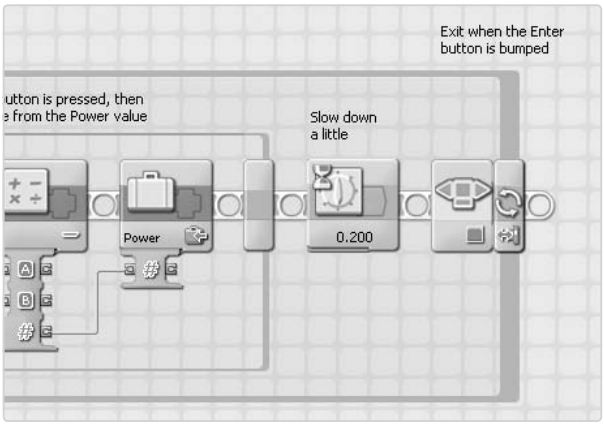


Figure 12-24: Adding the Wait Time block to the end of the loop body

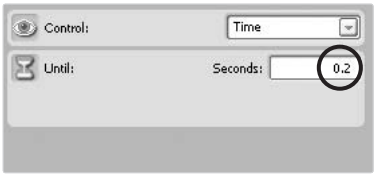


Figure 12-25: Pausing for 0.2 seconds

the display block

The Display block has four options for the Action setting: Text, Reset, Image, and Drawing. You're already familiar with the Text option. The Reset option simply clears the display and then shows the MINDSTORMS icon with the name of your program and the word *Running*. The Image and Display options require a little more explanation.

displaying an image

The Image option lets you display a picture on the screen. The MINDSTORMS software includes a wide variety of images to choose from, including several faces, arrows, and small animals. Figure 12-26 shows how the Configuration Panel looks when Image is selected as the Action setting. Select the image you want to use from the File list, and the image will be displayed in the preview area on the right side of the panel.

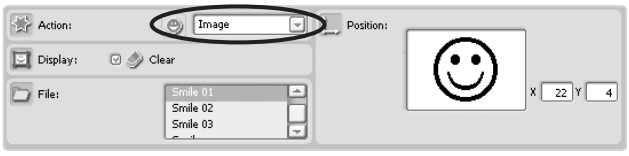


Figure 12-26: Displaying an image

The X and Y values on the far right of the Configuration Panel set the location of the lower-left corner of the image. Think of the display screen as a grid of dots called *pixels*. (*Pixel* is short for *picture element*.) The display is 100 pixels wide and 64 pixels high, and the location of each pixel is given by an X and Y value. The X value tells you the left-to-right location, with the values going from 0 to 99. The Y value tells you the bottom-to-top location, with 0 at the bottom of the display and 63 at the top (see Figure 12-27).

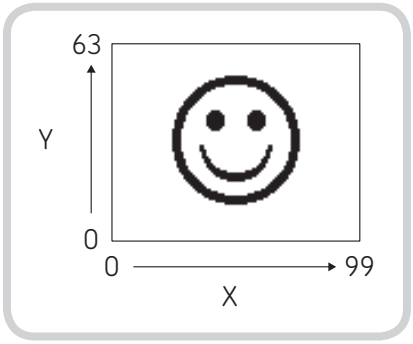


Figure 12-27: X and Y values for the NXT's display screen

Set the X and Y values by entering them in the boxes provided or by using your mouse to drag the image around the preview area. Depending on the values you choose for X and Y, part of the image may be cut off. For example, Figure 12-28 shows the Smile 01 image (also used in Figure 12-26) moved up and to the right so that only part of the image is displayed.

Figure 12-28: The image moved up and to the right

You can also use negative values for X and Y to move the image's starting point off the bottom or left side of the display, as shown in Figure 12-29. Unfortunately, if you enter negative values, the IDE will change them to zero. To use negative values, use your mouse to drag the image off the screen, or set the location using data wires.

Figure 12-29: The image moved down and to the left

power setting with images

In this section, you'll modify the PowerSetting program to use images instead of text to display the current Power setting by displaying an image of a snail on the left side of the screen, a rabbit on the right side, and an arrow to show the current setting, as shown in Figure 12-30. As you use the buttons to change the setting, the arrow should move left as you decrease the setting and right as you increase the setting. (Be sure to save your modified program with a new name so that you won't lose the original version.)

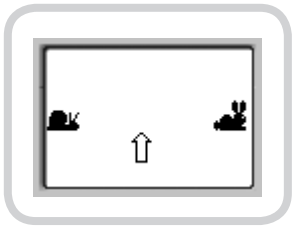


Figure 12-30: Using images to show the Power setting

The only part of the program you need to change is the section that displays the current value, as highlighted in Figure 12-31. To begin, delete the four existing blocks.

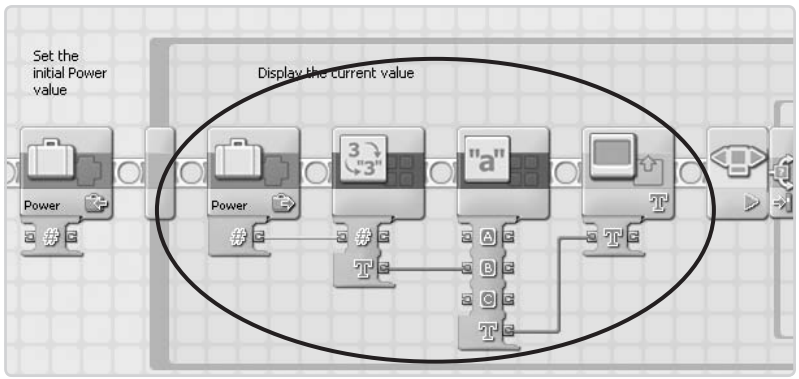


Figure 12-31: Replace these four blocks.

displaying the snail and rabbit

The snail and the rabbit will always be drawn in the same place on the screen, so add two Display blocks to draw them before displaying the arrow. Figure 12-32 shows the two new blocks, and Figures 12-33 and 12-34 show their Configuration Panels.

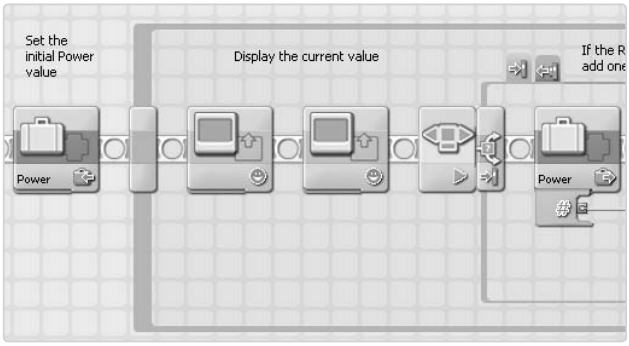


Figure 12-32: Displaying blocks to show the snail and rabbit

For both Display blocks, you need to select the file to use and set the location. The first Display block should use the Snail image, and the second should use the Rabbit image. Use 0 for the X setting to put the snail on the left side of the screen, and use 83 for the X setting to put the rabbit on the right side. I arrived at these settings by dragging the image to the side of the preview area with my mouse while watching how the X and Y values change. Use 25 for

the Y setting for both blocks so that the two images line up horizontally.

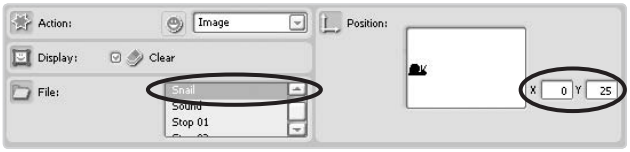


Figure 12-33: Showing the snail on the left side of the screen

Deselect the Clear option for the second Display block so that it doesn't erase the image from the first one. It's quite common to use several Display blocks together to display all the information you want (either images or text). The first Display block will usually have the Clear option checked so that you start with a clear screen, and the others will have the option unchecked so that they add to the information already displayed.

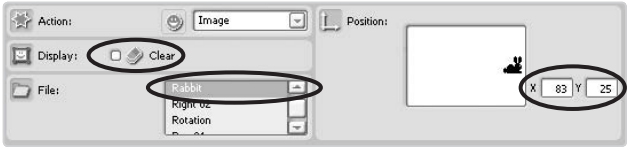


Figure 12-34: Showing the rabbit on the right side of the screen

display the arrow

Position the arrow between the snail and rabbit to indicate the current Power setting. Figure 12-35 shows the Configuration Panel for the Display block to show the arrow using the Forward image file. I found that setting the Y value to 9 puts the arrow just under the snail and the rabbit.

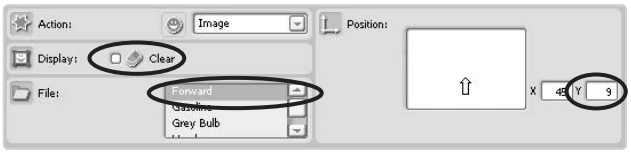


Figure 12-35: Showing the arrow

Control the left-to-right position of the arrow by setting the X value using a data wire, with the value based on the Power variable. The screen is 100 pixels wide, and the range for the Power setting is 0 to 100, so you can use the current value of the Power variable to position the arrow. However, you need to do just a little math to position the arrow correctly. The Display block's X setting controls the position of the left side of the image, not the center, but the value of the Power variable should match the position of the center of the arrow (not the left side).

When you first select the Forward image from the File list, the default X value is 45, and the arrow is centered left to right on the screen, as shown in Figure 12-35. When the Power variable is at 50, the arrow should be centered, so when the Power variable is 50, the Display block's X value should be set to 45. This means that you can calculate the correct X position by subtracting 5 from the Power variable.

Figure 12-36 shows the code to display the arrow, and Figures 12-37 and 12-38 show the Configuration Panels for the Variable and Math blocks. The Variable block reads the Power variable, and the Math block subtracts 5 from this value. The Math block's result is connected to the Display block's X data plug, which makes the X position of the arrow dependent on the Power variable. The position may change each time through the loop as you use the buttons to change the value. When the Power variable increases, the X value will also increase, and the arrow will move to the right; conversely, when the Power variable decreases, the X value will decrease, and the arrow will move left.

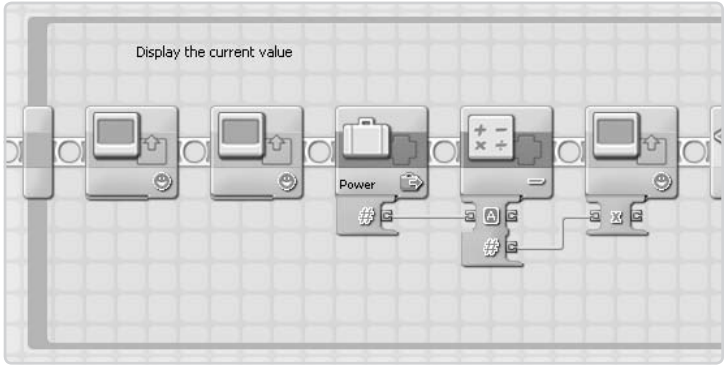


Figure 12-36: Using the Power variable to position the arrow

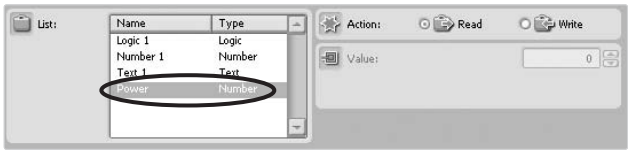


Figure 12-37: Reading the current Power value

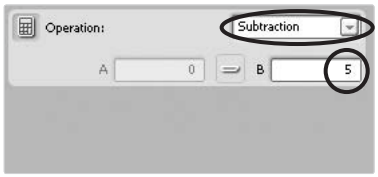


Figure 12-38: Subtracting 5 to set the position of the point of the arrow

When you run this program, the buttons affect the Power variable in just the same way they did for the original PowerSetting program. The only difference you should see relates to how the value is displayed: The Left button should move the arrow toward the snail, and the Right arrow should move the arrow toward the rabbit. The Enter button ends the program, just as in the original program.

drawing on the screen

The Display block's Drawing option lets you draw points (a single dot), circles, and lines. Figure 12-39 shows the Configuration Panel with the Point option selected. You can set the location of the point by entering the X and Y values or by clicking the desired location in the preview area.

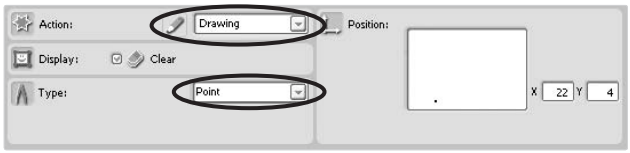


Figure 12-39: Drawing a point

Figure 12-40 shows the Configuration Panel for drawing a circle. In this case, the X and Y location marks the center of the circle. With Type set to Circle, an additional box to enter the radius of the circle is displayed, which lets you control the size of the circle.

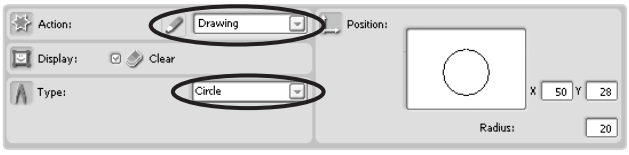


Figure 12-40: Drawing a circle

Figure 12-41 shows the Configuration Panel to draw a line. To draw a line, you need one point for each end of the line. The X and Y boxes set the location of one end of the line, and the End point X and Y boxes set the location of the other end.

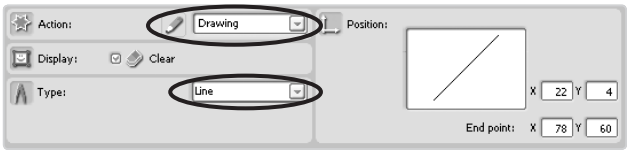


Figure 12-41: Drawing a line

the NXTSketch program

The NXTSketch program that you'll create in this section will use the line-drawing feature of the Display block to turn the TriBot into a sketch pad; the TriBot's two wheels control where the line is drawn. The program is fairly simple: It repeatedly draws a line from the last point used to the location defined by the current value of the two Rotation Sensors.

The NXTSketch program uses two variables, X and Y, to store the last location used. Both variables are initialized to zero at the beginning of the program. The Rotation Sensors are read in a loop to get the new location, with the B motor used for the new X value and the C motor used for the Y value. Finally, a line is drawn from the old location to the new location, and the values for the new location are stored in the X and Y variables to be used the next time through the loop.

In addition to drawing the line, there should be a way to clear the screen to start a new drawing. You can do this by setting the Clear option of the Display block if the Enter button on the NXT has been bumped. Listing 12-2 shows the pseudocode for this program.

```
set X to 0
set Y to 0
begin loop
   read the Rotation Sensor for motor B
   read the Rotation Sensor for motor C
   draw a line from X,Y to the point defined by the
      motor B and C positions. If the Enter button
      is pressed then set the Clear option.
   set X to the motor B position
   set Y to the motor C position
loop forever
```

Listing 12-2: The NXTSketch Program

defining the variables

First define the variables. Figure 12-42 shows how the X and Y variables are defined.

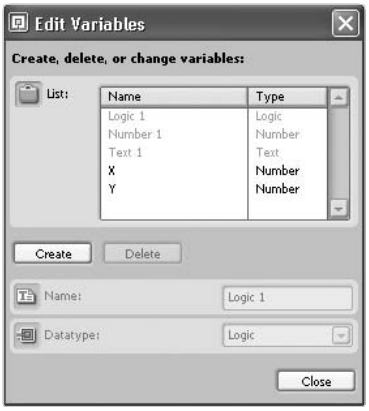


Figure 12-42: Defining the variables

initialization

This program is a little too long to show in one image, so I've broken it into three sections. Figure 12-43 shows the initialization of the two variables and the Rotation Sensors. Figures 12-44 through 12-47 show the Configuration Panels.

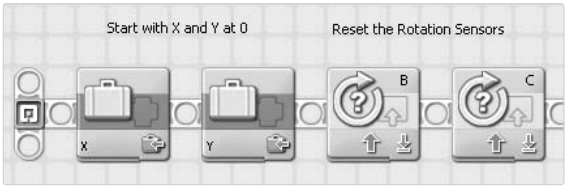


Figure 12-43: Initializing the variables and sensors

Both variables are initialized to 0.

Figure 12-44: Initializing X to 0

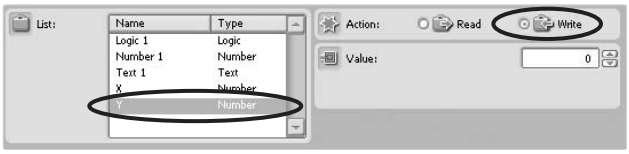


Figure 12-45: Initializing Y to 0

The Rotation Sensor blocks reset the sensors for the B and C motors.

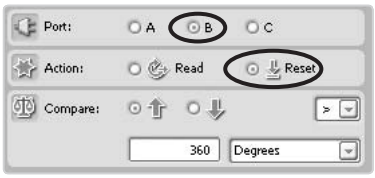


Figure 12-46: Resetting the Rotation Sensor for the B motor

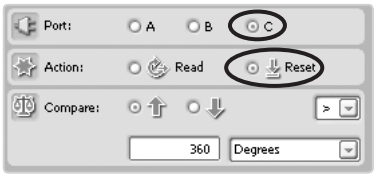


Figure 12-47: Resetting the Rotation Sensor for the C motor

drawing the line

Figure 12-48 shows the interesting part of the program, where the Display block draws a line. To draw a line, you need to give the Display block two points: one defined by the values of the X and Y variables and the other by values read from the Rotation Sensors for the B and C motors. All of these values are passed to the Display block using data wires.

The NXT Button block checks to see whether the Enter button has been bumped. The output value from this block will be true if the button has been bumped and false if it hasn't. This value is passed to the Display block and used to

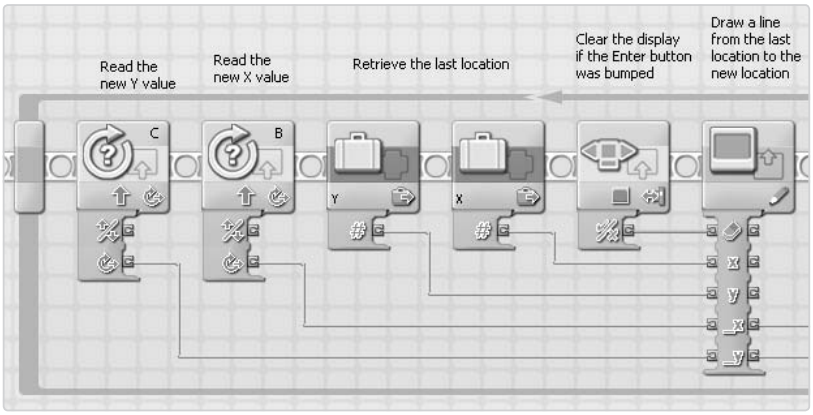


Figure 12-48: Reading the sensors and drawing the line

control the Clear option so that when you press the button, the Display block will clear the screen before drawing the line.

Notice how nicely the data wires are arranged in Figure 12-48. Five blocks supply the settings for the Display block. Because the order of these block doesn't matter, I was able to place them so that the data wires don't cross (of course, when I first put this program together, the wires were a bit of a mess). You can't always arrange the blocks to avoid crossing data wires, but when you can, it really helps to make a program easier to understand.

Figures 12-49 through 12-55 show the Configuration Panels for these blocks. The Loop block is set to loop forever.

Figure 12-49: Looping forever

The two Rotation Sensor blocks read the current motor position and pass the value to the Display block.

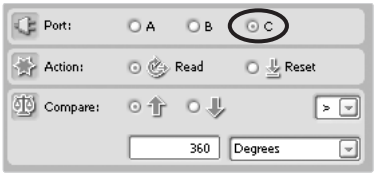


Figure 12-50: Reading the Rotation Sensor for the C motor

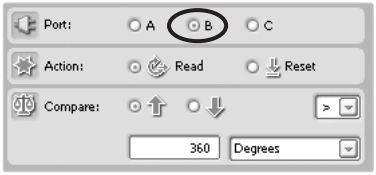


Figure 12-51: Reading the Rotation Sensor for the B motor

The Variable blocks read the current X and Y values and pass the values to the Display block.

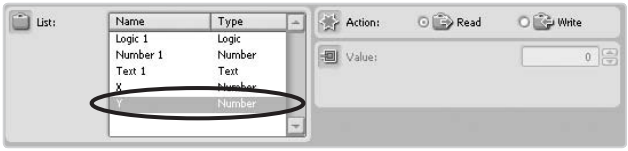


Figure 12-52: Reading the current Y value

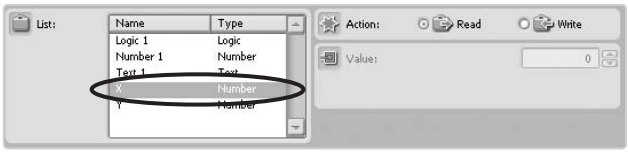


Figure 12-53: Reading the current X value

The NXT Button block will output true if the Enter button has been bumped and false if it hasn't. This value is passed to the Display block's Clear data plug.

Figure 12-54: Has the Enter button been bumped?

The Display block draws the line. The two points that define the line and the Clear option are all set using data wires, so you can ignore their Configuration Panel settings.

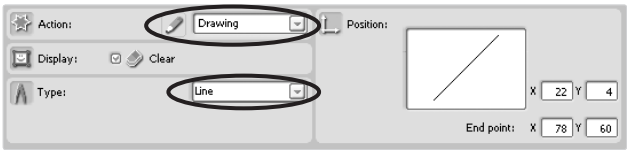


Figure 12-55: Draw the line. (The location and Clear value are set from data wires.)

saving the new location

The final section of code, shown in Figure 12-56, stores the values from the Rotation Sensors in the X and Y variables so that they can be used the next time the loop repeats. Figures 12-57 and 12-58 show the Configuration Panels for these blocks.

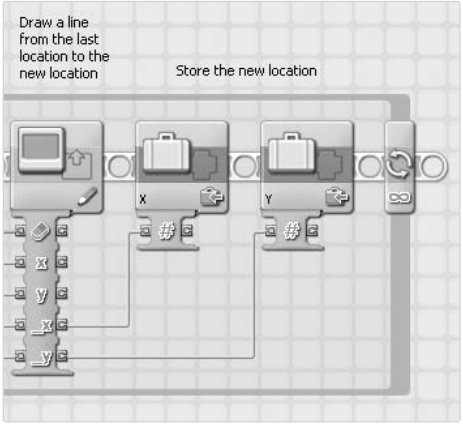


Figure 12-56: Storing the new values

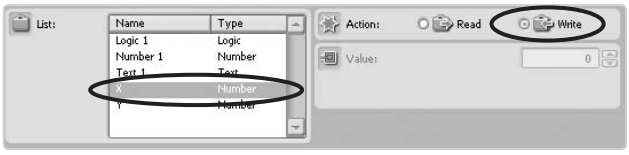


Figure 12-57: Storing the new X value

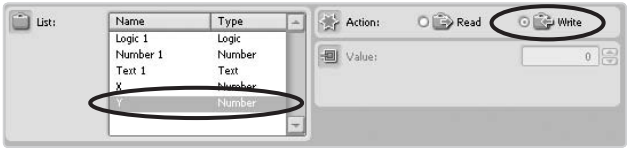


Figure 12-58: Storing the new Y value

testing the program

When you run the program, it should start with just a single dot in the lower-left corner of the screen. Create a drawing by turning the B wheel to move left to right and the C wheel to move top to bottom. Press the Enter button to erase the screen.

fixing the dials for NXT-G 2.0

If you're using NXT-G 2.0, you'll need to turn the wheels forward to start the drawing, and it may seem as if the control is backward. In NXT-G 2.0 the value from the Rotation Sensor will be negative if the wheel is turned backward, and the program won't draw anything if you move the wheel in the wrong direction. To make the Rotation Sensor block act like the 1.1 version, pass the value from the Rotation Sensor block through a Math block with the operation set to Absolute Value to make any negative values become positive values. Figure 12-59 shows the program with Math blocks added after each of the Rotation Sensor blocks. Figure 12-60 shows the Configuration Panel for the Math blocks (the two Match blocks are identical).

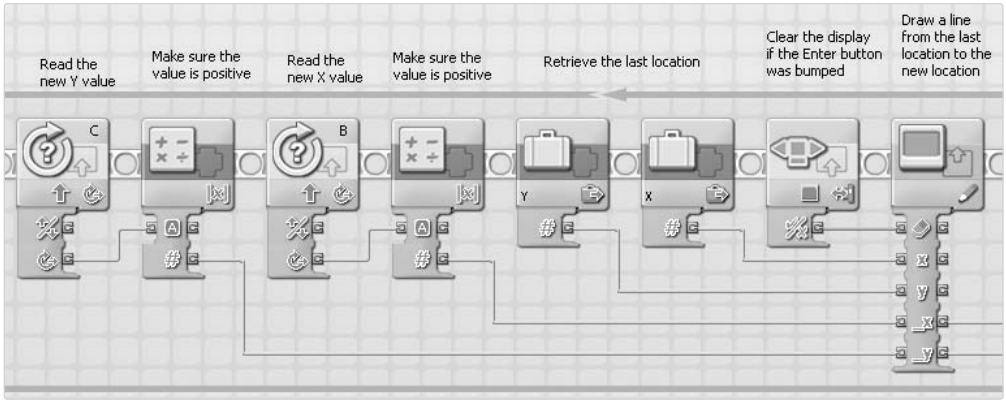


Figure 12-59: Avoiding negative Rotation Sensor values

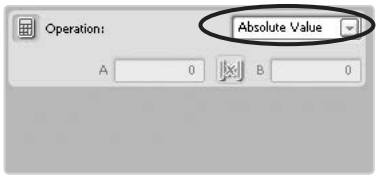


Figure 12-60: The result is the absolute value of the input.

With these two blocks added and the date wires connected, the program should now draw lines regardless of which way you turn the wheels.

conclusion

The NXT buttons give you a very convenient way to interact with your program. The buttons work with the program flow blocks just like the other sensors, and they operate much like a Touch Sensor. The PowerSetting program demonstrates how to use the Left and Right buttons to set the value for a variable.

The other programs in this chapter showed you how to use the Display block to do more than just print text. Using this block, you can display images and create drawings on the screen. Your programs can use these features to make your robot much more fun to interact with.

13

my blocks

A *My Block* is a block that you create from other blocks. In this chapter, you'll learn how to create My Blocks and how to use them in your programs. I'll walk you through the process of building three My Blocks, progressing from a very simple one that plays a chime to a more complex one that displays a number with a label. Along the way, you'll learn all you need to know to create your own blocks. You'll also learn how to keep your blocks organized and how to share them with other NXT users.

building bigger blocks

Creating a My Block is an easy way to group together and reuse a collection of blocks. By now you may have noticed that an NXT-G program can quickly become very large. Seemingly simple tasks, such as displaying a number or adding one to a variable, can require several blocks. Also, many of the same groups of blocks are used repeatedly in different programs and often within the same program. My Blocks help you cope with both of these issues.

creating a my block

I'll begin with a very simple example to show you how to create a My Block. Figure 13-1 shows an expanded version of the DoorChime program from Chapter 5 (the original program uses only two Sound blocks). You'll create a My Block from the four Sound blocks, which will make the program shorter and give you a Chime block that you can use in other programs.

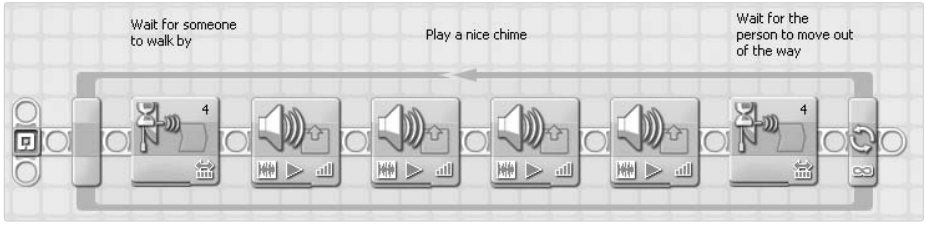


Figure 13-1: The DoorChime program

The first step is to select the blocks that you want to group together into a My Block. You can select the four Sound blocks by drawing a selecting rectangle around them (as shown in Figure 13-2) or by clicking the first block and then

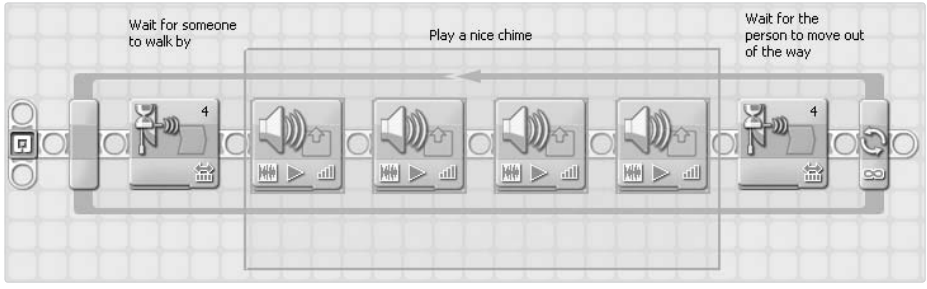


Figure 13-2: Selecting the four Sound blocks

holding down the SHIFT key while clicking the other three blocks.

1. Select the four Sound blocks.
2. To create a My Block from the selected blocks, click the **Create My Block** button on the toolbar, shown in Figure 13-3.

Figure 13-3: The Create My Block toolbar button

3. The My Block Builder window will appear, as shown in Figure 13-4.

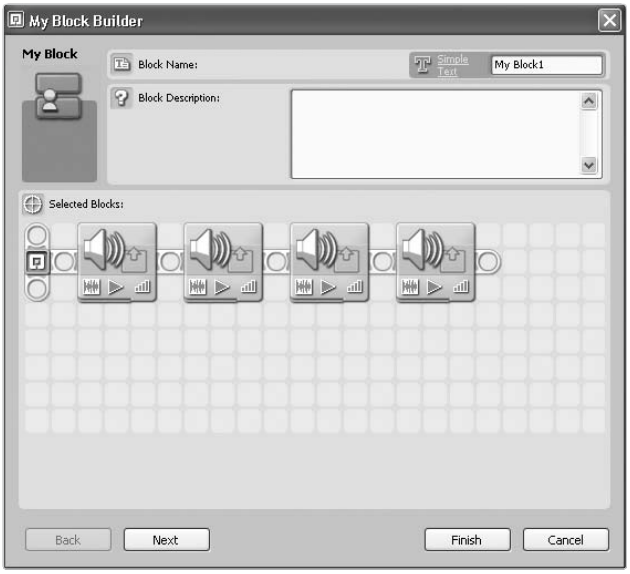


Figure 13-4: The My Block Builder window

The My Block Builder window contains boxes for entering a name and description for the new block. The Selected Blocks area lets you see which blocks were selected to make sure you're creating the block you expect. If the correct blocks aren't displayed, click the Cancel button and start again.

Follow these steps to enter a name and description for the block:

4. Enter **Chime** in the Block Name box.
5. Enter **Play a chime using several Sound blocks** in the Block Description box.
6. Click the **Next** button.

The My Block Builder window should now contain controls for building an icon for the new block, as shown in Figure 13-5. Create an new icon by dragging one or more icons from the lower section of the window into the box in the Icon Builder section. You can resize and move the individual images to create a new unique icon for your block. For the Chime block, I used three musical note icons, as shown in Figure 13-6.

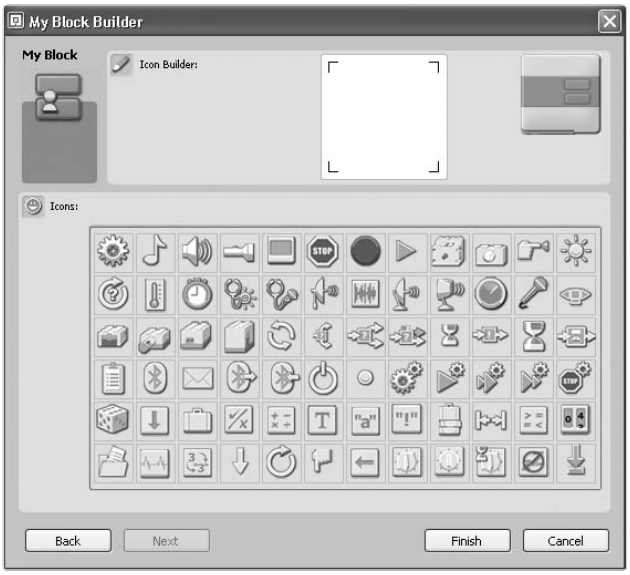


Figure 13-5: Building an icon with the My Block Builder window

Figure 13-6: Icon for the Chime block

7. Create an icon for the Chime block.
8. Click the **Finish** button.

Clicking Finish will create the Chime block and replace the four Sound blocks with the new block in the DoorChime program (as shown in Figure 13-7). After moving the blocks closer together, the program will look like Figure 13-8.

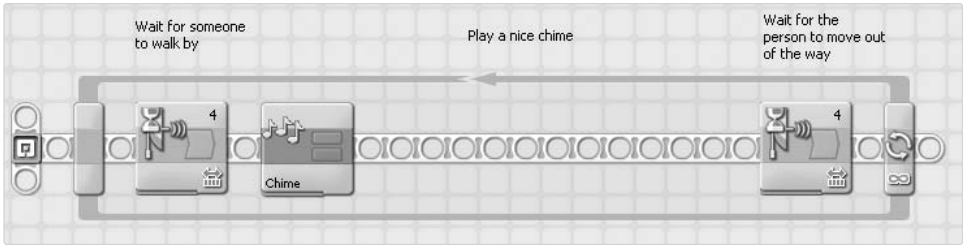


Figure 13-7: The DoorChime program after creating the Chime block

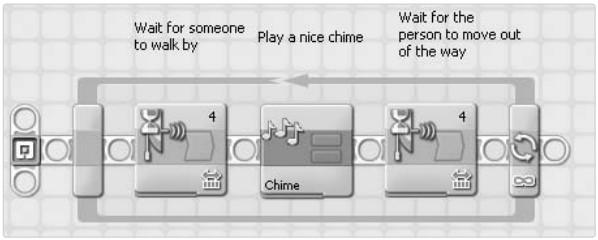


Figure 13-8: After moving the blocks closer together

The program is now smaller and simpler. It's easier to understand the purpose of a single block named *Chime* than four Sound blocks, which could be used for a variety of purposes. Although you've changed the arrangement of the blocks, you haven't changed the way the program works; when you run this program, it should behave exactly as it did before you created the Chime block.

the custom palette

Once you've created a My Block, you can use it in any program, just like any other block. All of your My Blocks will appear on the Custom Palette, which you open using the tab at the bottom of the Programming Palettes, as shown in Figure 13-9.

Figure 13-9: Tab for selecting the Custom Palette

Newly created My Blocks will appear in the top group on the Custom Palette, as shown in Figure 13-10. You can create new groups on the Custom Palette to organize your My Blocks, as explained in "Managing the Custom Palette" on page 174.

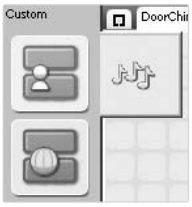


Figure 13-10: The Chime block on the Custom Palette

editing a my block

To edit a My Block, you can either double-click it or select it and choose Edit ▸ Edit Selected My Block from the menu. This will open the block in the MINDSTORMS software, where it will look like a small program. Now, to edit the Chime block, follow these steps:

1. Open the DoorChime program, and select the Chime block.
2. Choose **Edit ▸ Edit Selected My Block** from the menu.
3. The blocks that make up the Chime block should look like Figure 13-11.

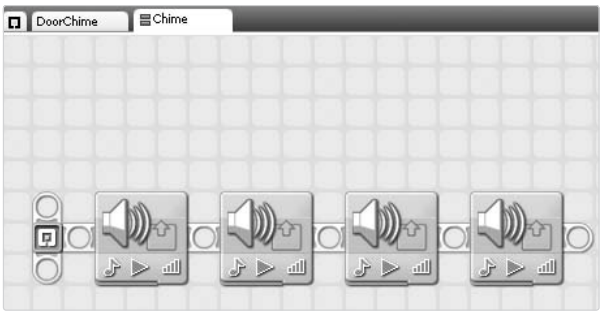


Figure 13-11: Editing the Chime block

Once the block is open, you can make changes, including adding and removing blocks, changing the settings of the blocks, or adding comments.

4. Change the Note settings for the four Sound blocks.
5. Save and close the Chime block (choose **File ▸ Save** and **File ▸ Close**).

Now when you download and run the DoorChime program, it should use the new settings for the Sound blocks. The new settings will also be used by any other program that uses the Chime block. In other words, when you edit a My Block, you affect every program that uses it. This can be a very good thing if you're fixing a bug, because you'll automatically fix the bug in all your programs. On the other hand, you need to make sure that the changes you make to improve one program won't adversely affect other programs.

configuring a my block

The Chime block is a little unusual in that it doesn't have any configuration settings; it always does the same thing. Most NXT-G blocks need some information to perform their function, and this will be true of most of the My Blocks you create.

For example, let's say you wanted to create a My Block from the Timer1 program from Chapter 11, shown in Figure 13-12. Recall that this is a programmable timer, meaning you can use a data wire to set the amount of time to wait. A My Block created from this program will be useful only if you can configure the length of the delay (otherwise it wouldn't be programmable).

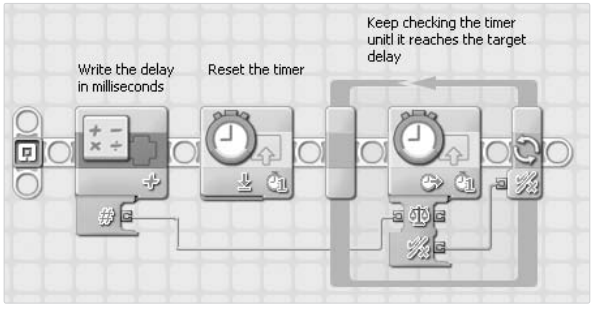
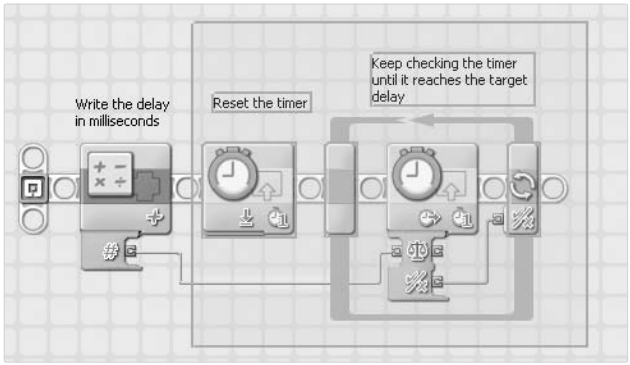


Figure 13-12: The Timer1 program

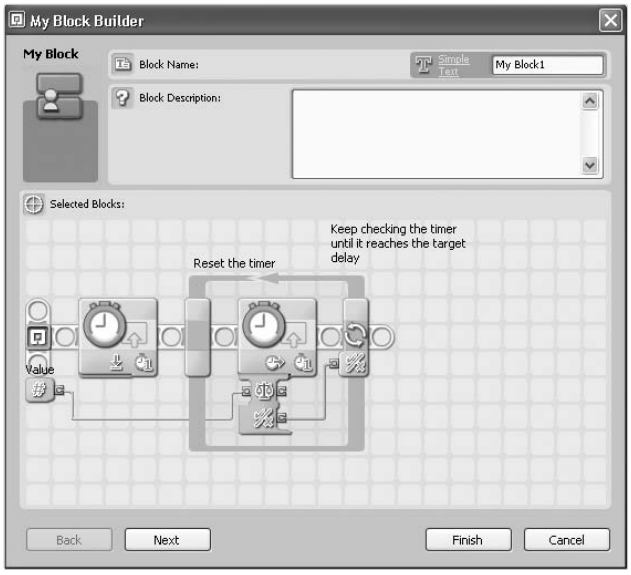
When you create a My Block, all the data wires that connect to the selected blocks will remain connected. The new My Block will have a data plug for any data wire that connects a selected block to an unselected block (one that doesn't become part of the My Block). To create a programmable timer block from the Timer1 program, select the Timer block and the Loop block but not the Math block. This will create a data plug on the new My Block, allowing you to set the delay.

Follow these steps to create the programmable timer block:

1. Open the Timer1 program if it isn't already open.
2. Select the Timer and Loop blocks, as shown here:

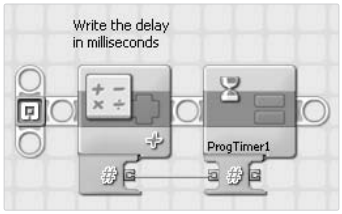


3. Click the **Create My Block** toolbar button. The My Block Builder window should look like this:



4. Enter **ProgTimer1** for the Block Name option and **A programmable timer using NXT timer 1** for the Description option.
5. Click the **Next** button.
6. Create an icon for the new block. I used the hourglass shape, as shown here:

7. Click the **Finish** button. The Timer1 program should now look like this:



The new ProgTimer1 block has a data plug to supply the delay, which is just what you need to make this block useful. However, to set the delay without using a data wire (perhaps while testing or debugging your program), you could also use the new block's Configuration Panel, as shown in Figure 13-13.

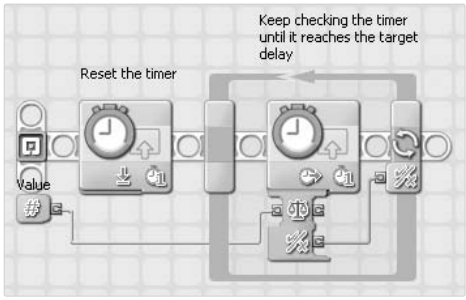
Figure 13-13: The ProgTimer1 block's Configuration Panel

A My Block's Configuration Panel will contain an item for each of the block's data plugs or for each data wire that was connected between the blocks used to create the My Block and the blocks in the original program. Depending on how the new block will be used, you may configure some of the settings using data wires and others using the Configuration Panel. You'll almost always set the delay for the ProgTimer1 block using a data wire, but you'll see examples of other My Blocks that are controlled mainly using the Configuration Panel.

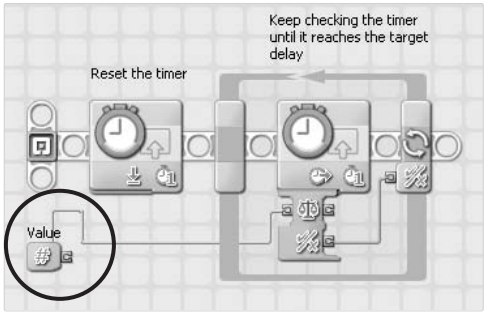
changing the name of a configuration item

The item in the ProgTimer1's Configuration Panel, as well as the data plug, is named *Value*. This isn't bad, but it could be more descriptive. The name used for a configuration item comes from a comment that appears just above the data plug for the item in the code for the My Block. To change the name, change the comment. For example, to change the name of the ProgTimer1's configuration item from *Value* to *Delay in milliseconds*, you would do this:

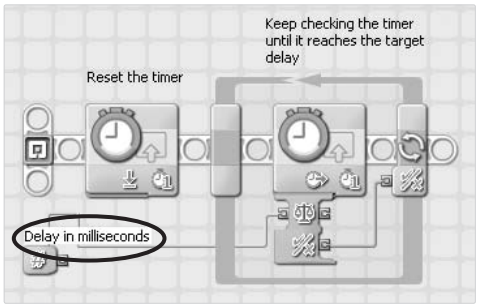
1. Open the Timer1 program if it isn't already open.
2. Double-click the ProgTimer1 block to open the code for the block:



The comment for the data plug is obscured by the Sequence Beam, and once you start changing the comment, it will extend over the Timer block. To make editing the comment easier, drag the data plug down a little with your mouse.



3. Double-click the comment that says *Value*.
4. Type **Delay in milliseconds**.



5. Save and close the ProgTimer1 block.

Now when you add the ProgTimer1 block to a program, the data plug and the item in the Configuration Panel will be named *Delay in milliseconds* instead of *Value* (as shown in Figure 13-14). For this example, changing the name is a small improvement. For My Blocks with several configuration items, changing the names from the default to something more meaningful will be a crucial step in creating useful blocks.

Figure 13-14: The updated Configuration Panel for the ProgTimer1 block

the DisplayNumber block

In this section, you'll build a My Block for displaying numbers on the NXT's screen. Several of the programs presented so far have used the Number to Text, Text, and Display blocks to display a labeled value. For example, Figure 13-15 shows the section of the PowerSetting program from Chapter 12 that displays the current value of the Power variable. In this section, you'll create the DisplayNumber block from the Number to Text, Text, and Display blocks, which will make displaying numbers much more convenient.

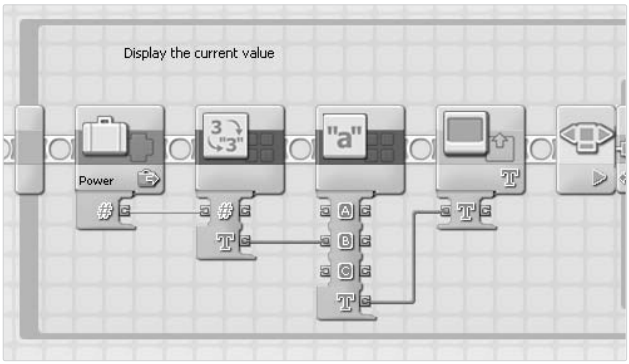


Figure 13-15: The Number to Text, Text, and Display blocks used to display a number

configuration items

Before creating a My Block, think about the configuration items that your new block will need. For example, here are the settings I typically use in the Number to Text, Text, and Display blocks:

* The number passed to the Number to Text block. This is the value to display.
* The A value for the Text block. This is the label for the value.
* The C value for the Text block. This value is used to print a unit after the value; for example, the SoundMachine program displays `Frequency 1256 Hz`. Not all values require a unit, so in other programs, this value has been blank.
* The Clear setting for the Display block.
* The Line number for the Display block to use.

To create a configuration item and data plug for each of these values in the DisplayNumber block, you need a data wire connected to the appropriate data plugs of the Number to Text, Text, and Display blocks. However, there is one small problem: The Display block doesn't have a data plug for setting the Line value. So before you can build the DisplayNumber block, you need to know how to control the Line setting using a data wire.

controlling the line setting using a data wire

The Display block's Configuration Panel (Figure 13-16) offers two ways to set the vertical location of the text: the Line and Y settings. Typically when displaying text, you use the Configuration Panel's Line setting (the Y setting is more often used when drawing or displaying an image). Since the

Display block doesn't have a data plug for the Line setting, you'll need to use the Y data plug instead. To do this successfully, you need to understand the relationship between these two settings.

Figure 13-16: The Display block's Configuration Panel

As you change the Line setting in the Configuration Panel, the Y setting will also change. In Figure 13-16, you can see that for a Line setting of 1, the Y setting is 56. Change the Line setting to 2, and the Y setting will change to 48. Change the Line setting to 3, and the Y setting will change to 40. Notice that each time you increase the Line setting by 1, the Y setting is reduced by 8.

The relationship between these two settings can be expressed as follows:

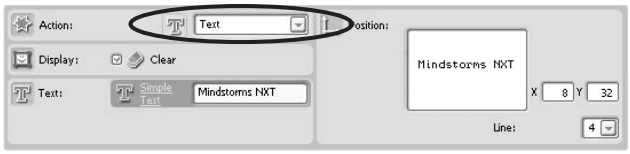
Y = 64 – (Line × 8)

To find the Y setting, multiply the desired Line value by 8, and then subtract that value from 64. In NXT-G, you can easily perform this computation using two Math blocks.

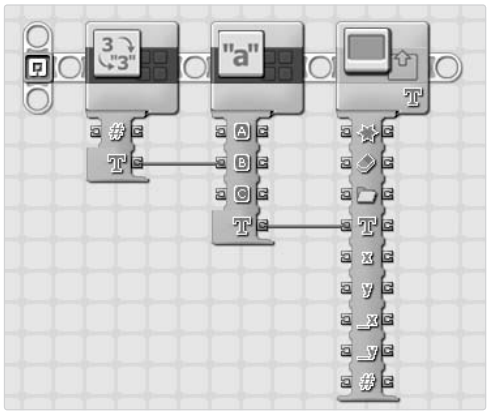
building the DisplayNumber block

Now you're ready to start building the DisplayNumber block. Begin with the familiar arrangement of the Number to Text, Text, and Display blocks, and then add the other blocks you need.

1. Create a new program named DisplayBlockBuilder.
2. Add a Number to Text block, a Text block, and a Display block.
3. Set the Display block's Action setting to **Text**.

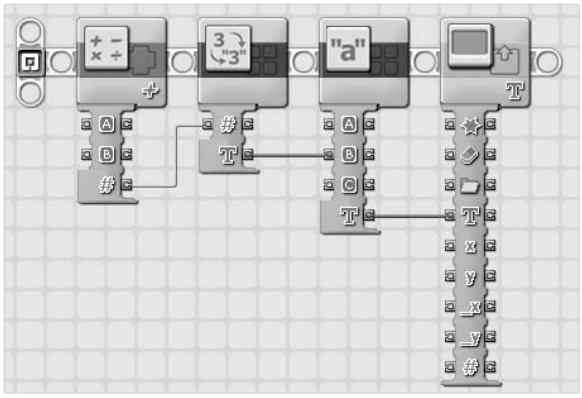


4. Open the data hubs of the Text and Display blocks.
5. Connect the Number to Text block's Text data plug to the Text block's B data plug.
6. Connect the Text block's Combined Text data plug to the Display block's Text data plug.



This is the familiar three-block pattern that you've used several times. Next add the blocks that supply the five configuration items: the value to display, the label, the unit, the Clear setting, and the Line setting. Begin with the value to display, which should be connected to the Number to Text block's Number data plug.

7. Add a Math block at the beginning of the program, to the left of the Number to Text block.
8. Connect the Math block's Result data plug to the Number to Text block's Number data plug.

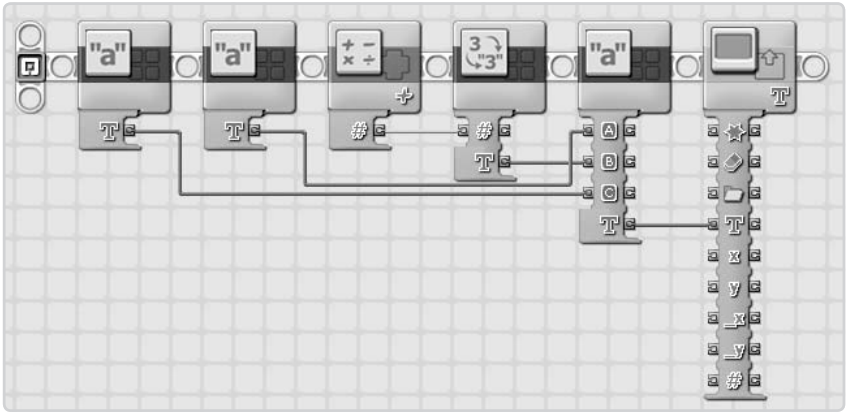


The only reason the Math block is there is to connect a data wire to the Number to Text block's Number data plug. The configuration of the Math block doesn't matter, and it won't become part of the DisplayNumber block. I used a Math block because we need to connect a data wire to

the Number to Text block's input data plug, and the Math block has a output data plug that uses a number. Any block that has an output data plug that uses a number would work just as well as a Math block.

To supply the label and unit to the Text block's A and C data plugs, use two Text blocks.

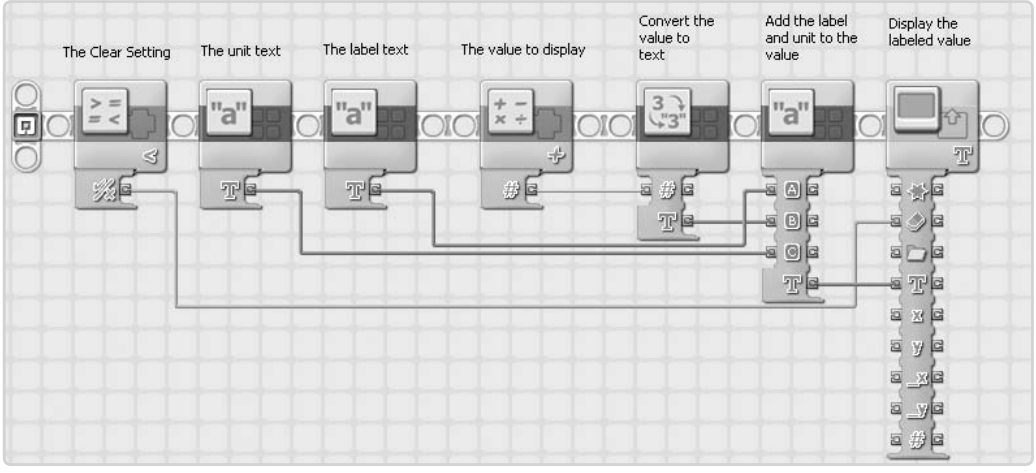
9. Add two Text blocks at the beginning of the program.
10. Connect the Combined Text data plug of one of the new Text blocks to the original Text block's A data plug.
11. Connect the Combined Text data plug of the other new Text block to the original Text block's C data plug.
12. Close the data hubs for the two new Text blocks and the Math block. The program should look like this:



Just as with the Math block, the configuration of these two Text blocks is not important. These two blocks are only there to connect data wires to the original Text block's A and C data plugs.

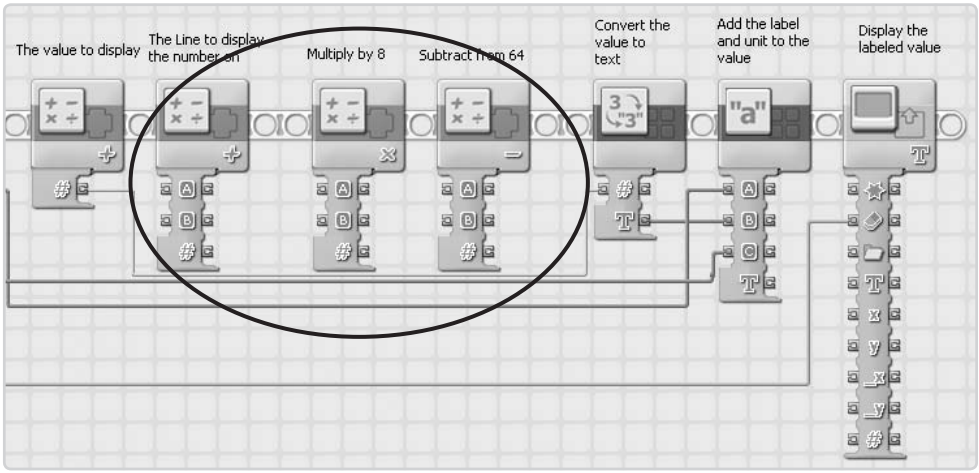
Next you'll add a Compare block in order to connect a data wire to the Display block's Clear data plug. I chose a Compare block because it has an output data plug that uses a logic value, which is the data type that the Clear data plug expects.

13. Add a Compare block at the beginning of the program.
14. Connect the Compare block's Result data plug to the Display block's Clear data plug.
15. Close the Compare block's data hub. The program should look like this (I've added some comments to make the purpose of each block clear):



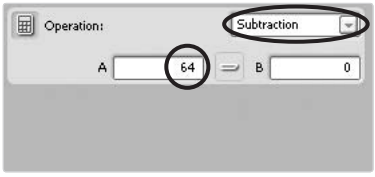
The last item you need to control is the Line setting, which is a little more complicated. You need three Math blocks, one to supply the Line value and two to convert from the Line value to the Display block's Y setting. To make it easier to select the blocks that will become the Display-Number block, place the three Math blocks just to the left of the Number to Text block.

Insert three Math blocks to the left of the Number to Text block. The three new blocks are highlighted here:

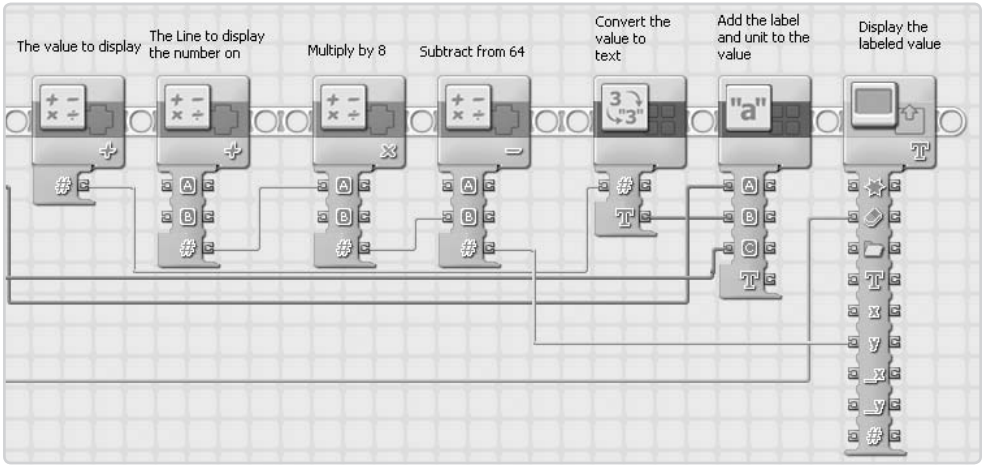


16. Select the second Math block (the middle one of the three you just added).
17. Set Operation to **Multiplication** and the B value to **8**. The Configuration Panel should look like this:

18. Select the third Math block.
19. Set Operation to **Subtraction** and the A value to **64**. The Configuration Panel should look like this:



20. Connect the first Math block's Result data plug to the second Math block's A data plug.
21. Connect the second Math block's Result data plug to the third Math block's B data plug.
22. Connect the third Math block's Result data plug to the Display block's Y data plug. This part of the program should now look like this:

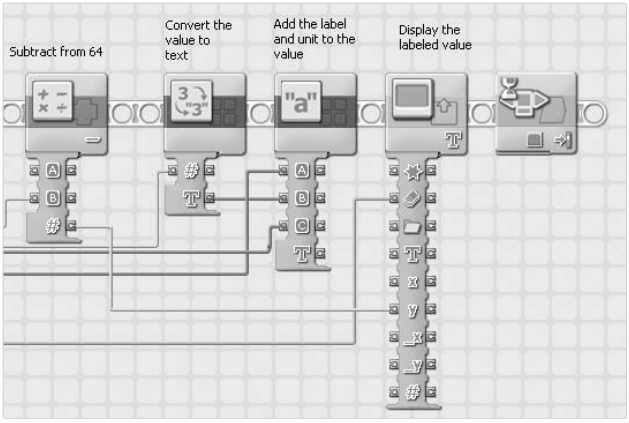


testing

All the blocks are now in place and connected. But how do you know it's really going to work? The only way to find out is to test the code, and the earlier you test it, the easier it will be to fix any problems. To test the code, you can set the five input values and then see whether the correct text is displayed. To see the text before the program ends, first add a Wait block to the end of the program and set it to wait until the Enter button is pressed.

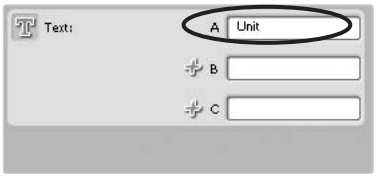
1. Add a Wait block to the end of the program.
2. Select **NXT Buttons** from the list of sensors. The Configuration Panel should look like this:

The end of the program should look like this:

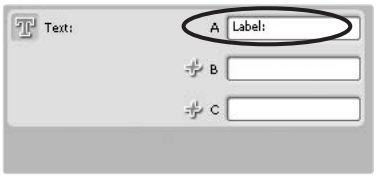


Now you'll configure the blocks that supply the input values. I'm going to skip the Compare block because you can't really test whether the Clear option is set properly when you're displaying only one value. Follow these steps to configure the program to display `Label: 17 Unit` on line 5:

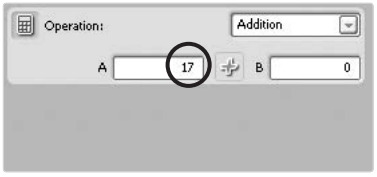
1. Select the first Text block. Set the A value to **Unit**, with a space before the *U*.



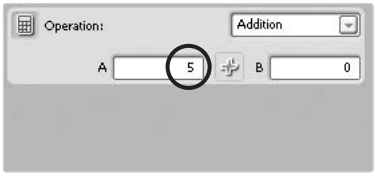
2. Select the second Text block. Set the A value to **Label:**, with a space after the colon.



3. Select the first Math block, the one that connects to the Number to Text block. Set the A value to **17**.



4. Select the second Math block, and set the A value to **5**.

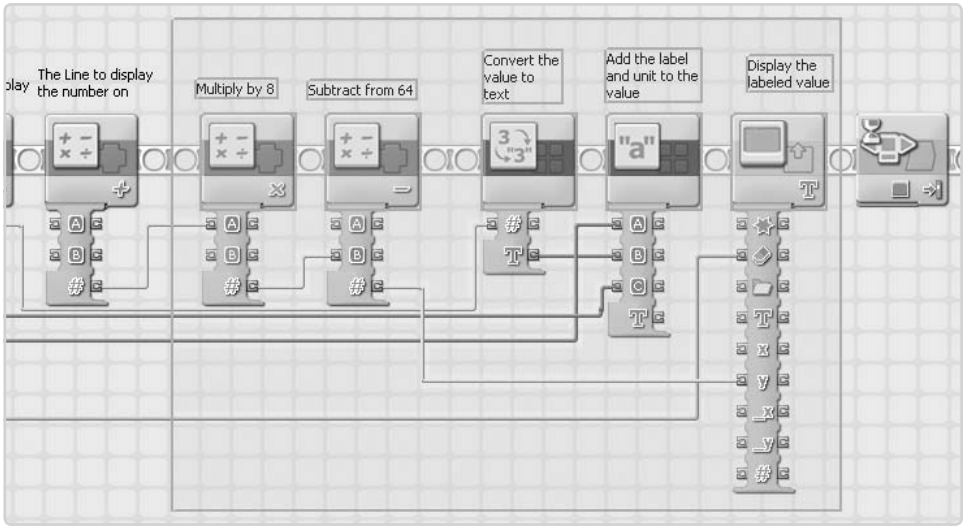


Download and run the program. `Label: 17 Unit` should be displayed near the middle of the screen. If the program doesn't work, check all the data wire connections; there are a lot of them, and it's easy to get one wrong.

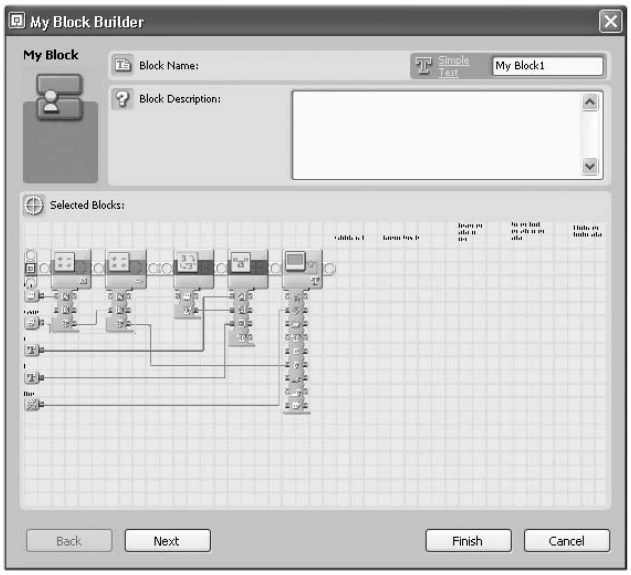
creating the DisplayNumber block

Once you have all the blocks in place, have all the data wires connected, and have some confidence that the block will work, it's time to create the My Block. Follow these steps to create the DisplayNumber block:

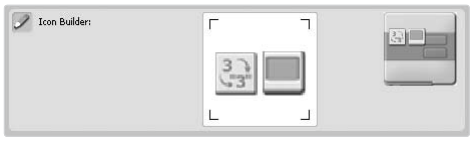
1. Select the two Math blocks that convert the Line value to the Display block's Y value, as well as the Number to Text, Text, and Display blocks, as shown here:



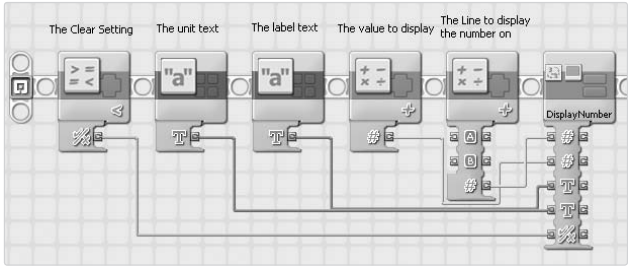
2. Click the **Create My Block** toolbar button. The My Block Builder window should be displayed:



3. Enter **DisplayNumber** for the Block Name option.
4. Enter **Display a number with a label and unit** for the Block Description option.
5. Click the **Next** button. The My Block Builder window should now show the controls for building an icon for the new block.
6. Create an icon for the new block. I mixed the icons for the Number to Text block and the Display blocks, as shown here:



7. Click the **Finish** button. The blocks that were selected should be replaced by a DisplayNumber block.



changing the names of the configuration items

Figure 13-17 shows the Configuration Panel for the new DisplayNumber block. As you can see, the names for most of the configuration items are not very meaningful. For example, the block uses two numbers: the value to display and the line number for the Display block to use. The Configuration Panel has boxes to enter two numbers, but the names *A* and *Number* don't tell you which value each box is used for.

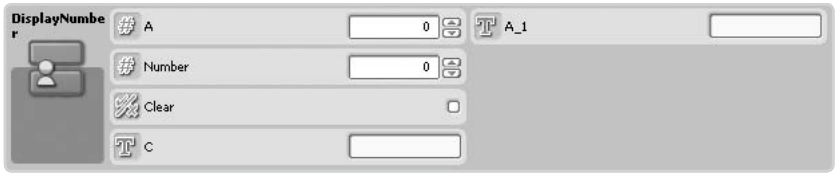
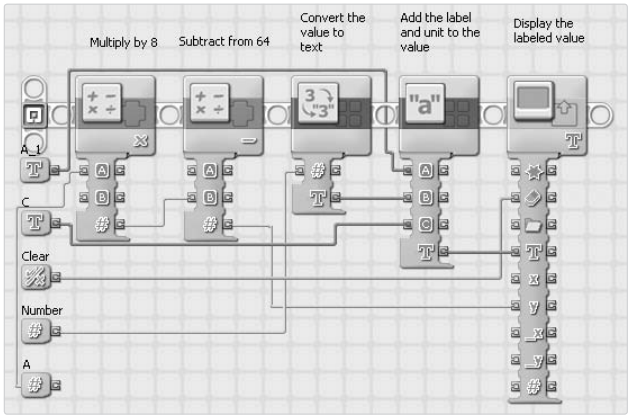


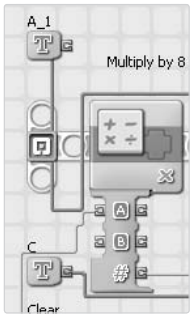
Figure 13-17: The Configuration Panel for the DisplayNumber block

To fix this problem, open the block, and change the names to something more meaningful by following these steps:

1. Double-click the DisplayNumber block to open it. It should look like this:

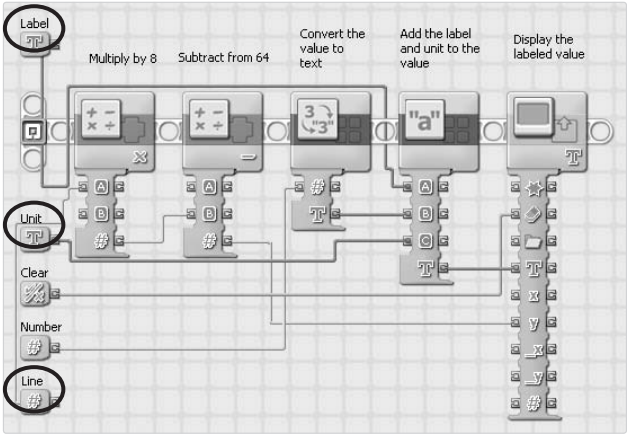


2. Move the data plug labeled *A_1* up so that you can see the comment easier.



By following the data wires, you can tell what each data plug is used for. The data plug labeled Clear is used to clear the display, and the one labeled Number is used for the value to display. These are reasonable names, so you don't need to change them. Follow these steps to change the comments for the other three data plugs:

3. Double-click the comment with the text *A_1*, and change it to **Label**.
4. Double-click the comment with the text *C*, and change it to **Unit**.
5. Double-click the comment with the text *A*, and change it to **Line**.



6. Save and close the DisplayNumber block.

The Configuration Panel won't look any different in the DisplayNumberBuilder program, and it will continue to use the original names. To see the updated Configuration Panel (shown in Figure 13-18), add the DisplayNumber block to a new program. The DisplayNumber block should appear on the Custom Palette, as shown in Figure 13-19.

Figure 13-18: The updated Configuration Panel for the DisplayNumber block

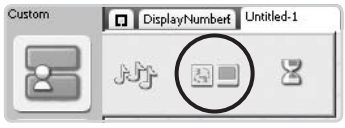
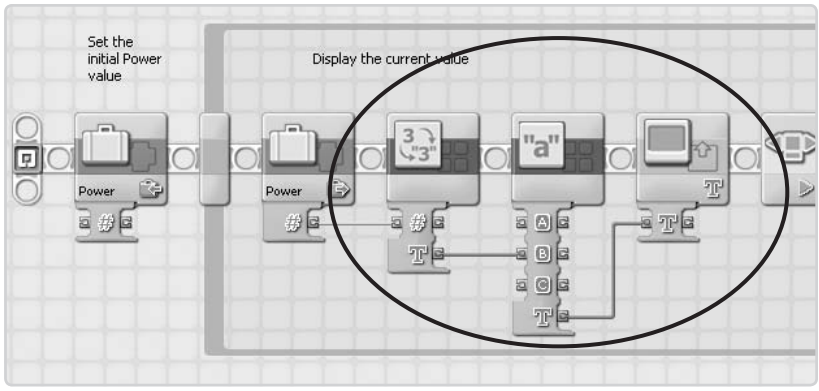


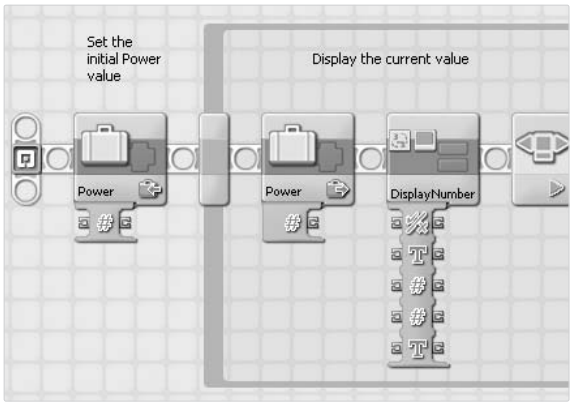
Figure 13-19: The DisplayNumber block on the Custom Palette

using the DisplayNumber block

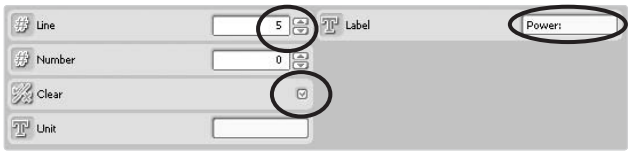
Now that you've created the DisplayNumber block, let's put it to use. Follow these steps to use the DisplayNumber block in the PowerSetting program, replacing the Number to Text, Text, and Display blocks:

1. Open the PowerSetting program.
2. Delete the Number to Text, Text, and Display blocks highlighted here:
3. Add a DisplayNumber block (from the Complete Palette) after the Variable block.

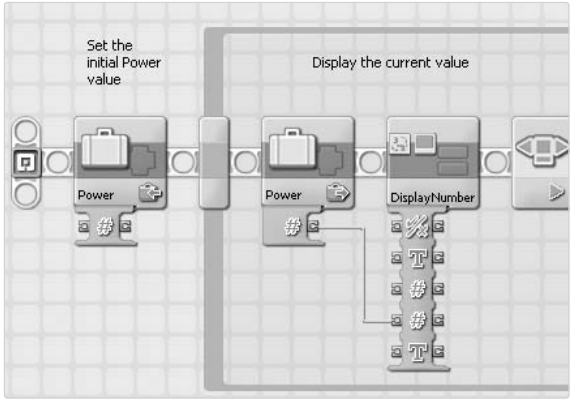




4. In the DisplayNumber block's Configuration Panel, set the Line item to **5** and the Label item to **Power:** (with a space after the colon).
5. Check the Clear option.



6. Draw a data wire from the Variable block's Value data plug to the DisplayNumber block's Number data plug. The Number data plug is the second one from the bottom, which you can tell by holding the mouse cursor over each data plug and reading the tool tip.



Download and run the program, and it should behave just like the original. The benefit of using the DisplayNumber block is that it makes the program smaller and easier to understand. We'll use the DisplayNumber block in several programs in the coming chapters.

managing the custom palette

After creating several My Blocks, you'll eventually want to delete some of them or change how the blocks are arranged on the Custom Palette. Each My Block is stored in a file on your computer, just like your NXT programs. The arrangement of My Blocks on the Custom Palette is controlled by the arrangement of folders and the My Block files on you computer.

Selecting the Edit ▸ Manage Custom Palette menu item will open the *Blocks* folder, which contains the items on the Custom Palette, as shown in Figure 13-20. Each subfolder listed here will be shown as a group on the Custom Palette. The My Block files within each folder will be shown as the blocks within each group. For example, the My Blocks subfolder (shown in Figure 13-21) contains the files for the three blocks you've created.

You work with these folders and files the same way you work with other files on your computer. To delete a My Block, simply delete the file. You can also rename a My Block by renaming the file.

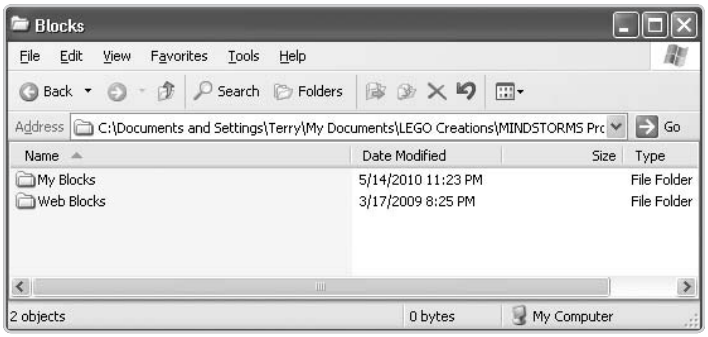


Figure 13-20: The folders corresponding to the groups on the Custom Palette

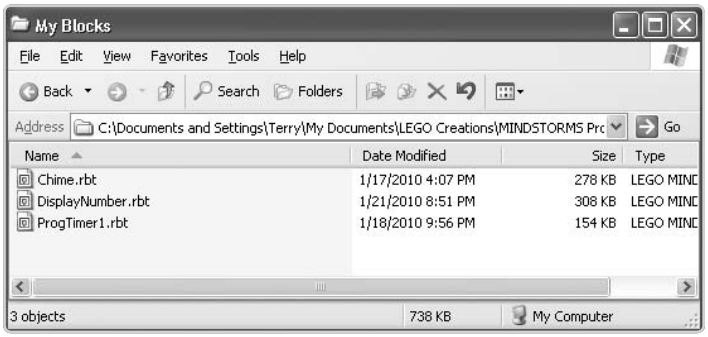


Figure 13-21: The files containing the My Blocks

NOTE **Deleting or renaming a My Block will break any program that uses the block. After renaming a My Block, you'll need to edit any program that uses it, replacing the block that uses the old name with the newly renamed block.**

You can also create new folders to help keep your My Blocks organized. For example, you could create a *Timers* folder to hold My Blocks created from the three programmable timers from Chapter 9, as shown in Figure 13-22. When you create the new blocks, they're placed in the *My Blocks* folder. After creating the *Timers* folder in the *Blocks* folder (the one that contains the *My Blocks* folder), you can move the three files from the *My Blocks* folder to the new *Timers* folder. The Custom Palette will then

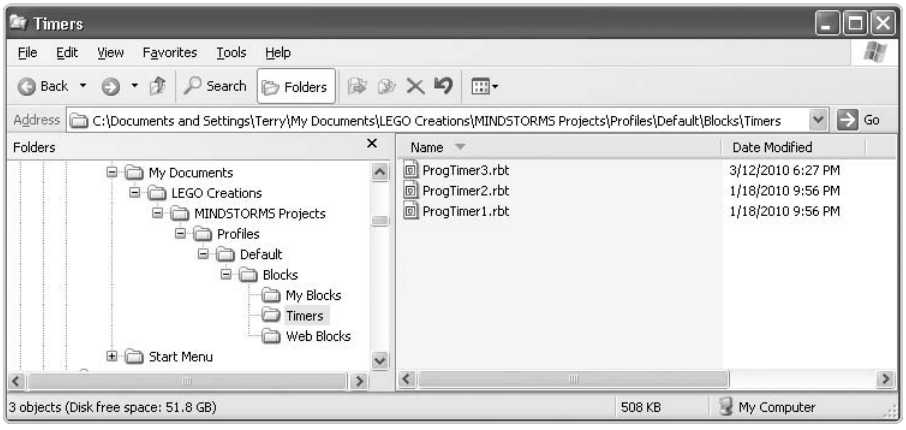


Figure 13-22: The Timer *folder containing the three programmable timers*

contain a Timers group containing the three programmable timer blocks, as shown in Figure 13-23.

NOTE **Moving a file to a different folder will break any program that's using the block, just as renaming a file does. After reorganizing your blocks in this way, you'll need to edit any programs that use the blocks.**

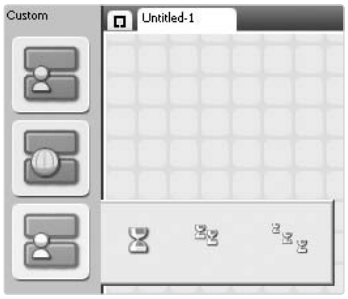


Figure 13-23: The new Timers *group on the Custom Palette*

sharing programs with my blocks

To share a program that uses My Blocks, you have to include the My Block files as well as the main program. You can do this in two ways, depending on which version of the MINDSTORMS software you are using.

copying files

With any version of the MINDSTORMS software, you can share a program by copying all the files it needs. Although you can place the file for the main program anywhere you want, the files for any My Blocks need to be in the same folders as they were on the original computer.

Start by copying the main program file, and then select Edit ▸ Manage Custom Palette to open the *Blocks* folder. Find and copy all the block files that the program needs to the email message, flash drive, floppy disk, or other medium you are using to share the files. On the destination computer, first copy the main program file. Then select Edit ▸ Manage Custom Palette to open the *Blocks* folder, and copy all the block files to the same place they were on the original computer, either in the *My Blocks* folder or in another folder you created (for example, the *Timers* folder).

create pack and go

NXT-G 2.0 includes the new Create Pack and Go feature, which creates a single file containing all the files used by a program. After creating your program, select Tools ▸ Create Pack and Go to display the Create Pack and Go window. Figure 13-24 shows how this window looks for the DoorChime program. The new file *DoorChime.rbtx* will contain both the DoorChime program file (*DoorChime.rbt*) and the Chime My Block file (*Chime.rbt*). Opening the *DoorChime.rbtx* file on another computer will copy the Chime My Block file to the correct location and open the DoorChime program in the MINDSTORMS software.

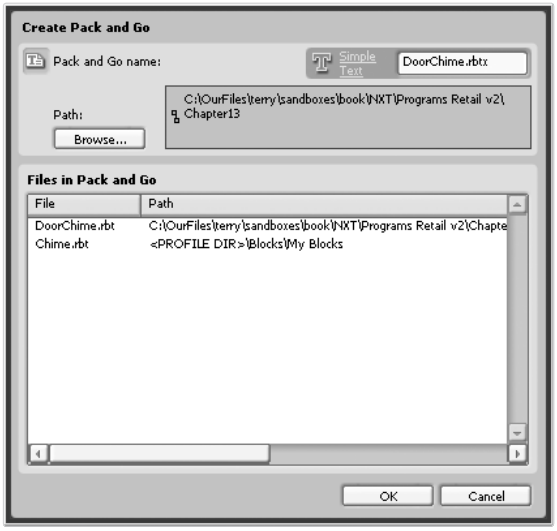


Figure 13-24: Creating a package for the DoorChime program

advanced my block topics

After creating the Chime, ProgTimer1, and DisplayNumber blocks, you should understand how My Blocks work and be able to create blocks for your own programs. The examples presented here are typical of the kinds of blocks you'll want to create for yourself. However, before ending this chapter, I'll cover a few more aspects of My Blocks that you should be familiar with. The following sections deal with using variables with My Blocks, nesting My Blocks, dealing with broken My Blocks, and adding data plugs to a My Block.

variables and my blocks

You can use variables in your My Blocks just as you do in the main program. The important thing to know is that a program and all the My Blocks it includes share the same list of variables. If you define a variable with the same name and data type in both the main program and in a My Block, the variable will be shared between the them. You can set a value in the main program and then use the value in a My Block, and any changes you make to the variable within a My Block will also be seen by the main program.

You can use variables to share information between the main program and the My Blocks it uses or between two (or more) My Blocks. For example, if you split the WallFollower program into three My Blocks, you could use one variable to control the Power setting of all the Move blocks, even though the Move blocks will be divided among the three My Blocks.

Variables are also useful if you need a My Block to remember a value. For example, you could write a My Block named DisplayNextLine to display scrolling text on the NXT's screen. The first time the block is used, it will clear the screen and display the text on line 1. The next time the block is used, it will display the text on line 2, the next time on line 3, and so on, until it displays the text on line 8. The next time it's used, it will clear the screen and display the text on line 1. To accomplish this, the block needs to use a variable to remember which line it used so that it can use the next line when the block is used again.

There is a potential for bugs caused by this sharing of variables. Accidentally using the same variable between the main program and a My Block, or between two My Blocks, can wreak havoc on an otherwise well-constructed program. Choose your variable names carefully to avoid this issue. For example, when creating the variable for the DisplayNextLine block described earlier, you could use the name *DNL_Line*. Starting the variable name with an abbreviation for the block name results in a name that is unlikely to be accidentally used in another program or My Block.

nesting my blocks

A My Block can contain other My Blocks, allowing you to build larger and more complicated blocks out of smaller and simpler ones. Instead of having one really large program, you can divide the functionality into a few logical pieces. For example, the WallFollower program from Chapter 7 can be divided into three My Blocks; one that follows the wall, one that turns at a corner, and one that turns into an opening, as shown in Figure 13-25.

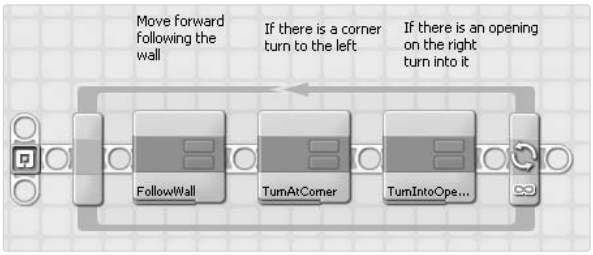


Figure 13-25: The WallFollower program split into My Blocks

It's easy to tell what this program does from a high level, and you can open the My Blocks to see how each individual part works. Each of these three My Blocks may use some smaller My Blocks.

The ability to nest My Blocks also makes it possible to use some very small, special-purpose blocks. For example, the DisplayNumber block uses two Math blocks to convert the Line value into the Y value used by the Display block. You could create a DisplayLineToY block from these two Math blocks and then reuse it for other blocks that display values, like the DisplayNextLine block described in the previous section. Although this block is very small, using it can help avoid some math errors that are very easy to make.

broken my blocks

A block will appear broken (as shown in Figure 13-26) if the MINDSTORMS software can't find the block file. This can happen if you delete, rename, or move a My Block file or if you copy a program from a friend and neglect to copy the My Blocks the program uses.

Figure 13-26: A broken block

A block will also appear broken if it has two data plugs with the same name or if there is some other problem with the data wire connections. If your program has a broken block, try opening the block. You'll see an error message if the MINDSTORMS software can't find the block file. If you can successfully open the block, then check the names of each data plug and the connections of the data wires.

adding a data plug

You can edit a My Block to add new blocks, remove blocks, or change the settings of the blocks. However, you can't add a new data plug after a My Block has been created. If you want to add a new data plug, you must re-create the My Block. For this reason, it's a good idea to save a copy of the program you're using to create the My Block, just before creating the new block.

For example, you could save a copy of the DisplayNumberBuilder program after testing it but before actually creating the block. Then if you later decide to add another data plug, maybe to control the horizontal position of the text, you could start with the saved program. This is much easier than re-creating the block all over again from the beginning.

conclusion

Creating your own blocks is a simple yet powerful way to reuse code, making your programs easier to understand and less prone to error. The three My Blocks presented in this chapter show how to create My Blocks with varying degrees of complexity. Simple blocks such as the Chime block allow you to easily reuse code and help keep programs to a manageable size. Complex blocks with many configuration options such as the DisplayNumber block can help reduce the problems associated with rewriting the same complicated code several times.

14

math and logic

Working with numbers is an important part of many programs. In this chapter I'll explain how NXT-G works with numbers in order to help you use the Math block successfully. I'll also discuss how to use the Logic and Range blocks to expand the types of decisions your programs can make.

computer math

Computers have a well-deserved reputation for being very good at math. In NXT-G, the Math block is used to perform any calculation your program needs. This block is very easy to use; you simply select the operation to perform and provide the numbers to work with using either the Configuration Panel or data wires. Performing complex calculations is a simple matter of combining several Math blocks.

Although the Math block is easy to use, you need to be aware of some issues when using it. Computers don't perform math exactly the way people do. Because of the way computers work, there are limitations to the types of calculations that they can perform correctly. Most of the time the Math block will give you the same answer you would get if you performed the operation using pencil and paper, but not always. Knowing the Math block's limitations will help you avoid subtle errors when it doesn't behave the way you might expect.

The way numbers work in NXT-G changed between version 1.1 and version 2.0. The original NXT-G software uses integer math, which uses only positive and negative whole numbers. On the other hand, NXT-G 2.0 uses floating-point math, which allows fractional values. In the following sections, I'll discuss the limitations of each approach.

integer math

In NXT-G 1.1, all numbers are integers, such as 27, 134, or -28. You can use positive or negative numbers, but no decimals. Working with integer math is fairly simple and comes naturally to most people. The two things you need to know when using integer math with NXT-G are the range of values supported and the way division works.

range of values

Most computers work with only a limited range of values. In NXT-G, the range of values you can use is from -2147483648 to 2147483647, which is slightly more than 2 billion for either positive or negative values. This is quite a large range and is more than sufficient for most purposes. As long as you keep your values within this range, the results from the Math block will be correct. If you go outside this range, for example if you multiply 1 billion by 4, the result will be a large negative number, which is incorrect.

division

The other limitation of integer math involves division. Working with integers isn't a problem for addition, subtraction, and multiplication. If you take any two integers and add, subtract, or multiply them, you'll always get an integer for the result. However, the same is not true for division.

When you divide two numbers using integer math, the result will be rounded down to an integer, meaning the fractional part is cut off. For example, if you divide 5 by 2, the answer will be 2.5, but when the Math block divides these two numbers, its result will be 2, because it can't handle the .5. It's important to note that the fractional part is simply dropped instead of being rounded to the closest integer. When the Math block divides 399 by 100, it will give 3 as the result, even though the real value is much closer to 4.

Any time you use division as part of a calculation, the final result can be inaccurate. What's important is the size of the error: the difference between the real result and the result you get using integer math. As long as the error is small, most programs won't have a problem. For example, if you're calculating the Power setting to use for a Move block and you use 74 instead of 75, you probably won't notice. The following sections give you some ways to avoid large errors when using division.

order of operations

You now know that when you divide two numbers, the result will be rounded down to an integer. The error will always be less than one, which is really not too bad. However, if you then use the result as part of other calculations, the final result can have a larger error. For example, say you want to compute 9 ÷ 4 × 10. When you divide 9 by 4 by hand, the result will be 2.25, and when using integer math, the result will be 2. In this case, the error is 0.25, which is not too bad. But now when you multiply the result (2) by 10, you'll get 20. If you do the same operations by hand (or with a calculator), the result is 22.5, which means the error has grown from 0.25 to 2.5.

To avoid this problem, perform the division last whenever possible. For example, you can rearrange the previous expression to be 9 × 10 ÷ 4. Now, because the multiplication is done before the division, the result will be 22, and the error is only 0.5. Table 14-1 summarizes these results.

table 14-1: the effect of the order of operations on the error

expression	real result	result using integer math	error
9 ÷ 4 x 10	22.5	20	2.5
9 x 10 ÷ 4	22.5	22	0.5

scaling values

In the previous example, the numbers we started with (9, 4, and 10) are all integers, but sometimes you'll need to work with numbers that are not integers. For example, to convert from inches to centimeters, you need to multiply by 2.54. However, 2.54 isn't an integer, so you can't use that value with the Math block, and using either 2 or 3 (the closest integers) could result in a substantial error.

One way to deal with this is to scale the value up; that is, multiply it by another number so that the fractional part is not lost. For example, if you scale 2.54 up by 100, you get 254. Of course, after using 254 in the calculation, you'll need to scale the final value down by 100 to get the correct result. So to convert 25 inches to centimeters, you could multiply 25 by 254 and then divide the result by 100. This will give you a more accurate result than multiplying 25 by either 2 or 3. Table 14-2 compares the results of converting 25 inches to centimeters, with and without scaling.

table 14-2: the effect of scaling on the error

	expression	real result	result using integer math	error
Without scaling	25 × 3	63.5	75	11.5
With scaling	25 × 254 ÷ 100	63.5	63	0.5

The scaling technique is used in the Odometer program presented in the next section to display how far the TriBot has moved based on the reading from the Rotation Sensor.

odometer

The Odometer program presented in this section reads the Rotation Sensor for motor B and converts the value to a distance measurement. Aside from the math, this program is very simple; it continually reads the Rotation Sensor, converts the value to a distance, and displays the result.

To convert the Rotation Sensor reading from degrees to a distance measurement, the program uses a variable, Wheel Circumference, which is the distance the robot travels in one rotation of the wheel. You can use either inches or centimeters when setting the wheel circumference value. The circumference of the wheel is 5.25 inches, or 13.33 cm. Figure 14-1 shows the Edit Variable dialog with this variable defined.

NOTE **If you're using the balloon tires, use 6.9 inches or 17.78 cm for the wheel circumference.**

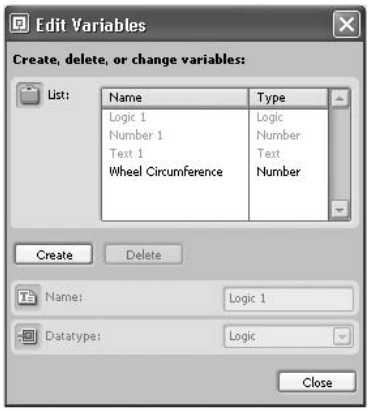


Figure 14-1: Defining the Wheel Circumference variable

The reading from a Rotation Sensor block will be in degrees. The simple approach to the conversion from degrees to inches or centimeters is as follows: Divide the number of degrees by 360 to get the number of rotations, and then multiply the result by the wheel circumference to get the distance traveled. Written as an expression, this would look like this:

degrees ÷ 360 × wheel circumference

Of course, this approach suffers from both problems mentioned earlier (otherwise it wouldn't be a very good example). You can get a more accurate result by making a few changes:

1. Scale the wheel circumference value up by 100 (use either 525 for inches or 1,333 for centimeters).
2. Multiply the degrees by the wheel circumference first, and then divide by 360.
3. To match the change to the wheel circumference value, the final result needs to be divided by 100.

With these changes, the expression becomes the following:

degrees × wheel circumference ÷ 360 ÷ 100

This expression makes up the main part of the Odometer program, shown in Figure 14-2.

NOTE **You could combine the two Math blocks that divide the input by 360 and 100 with one block that divides by 36,000. I prefer to use two blocks because I think of this as two separate operations, but that's just my preference.**

The program starts by setting the scaled Wheel Circumference variable and putting the value on a data wire. Figures 14-3 through 14-4 show the Configuration Panels for the two Variable blocks.

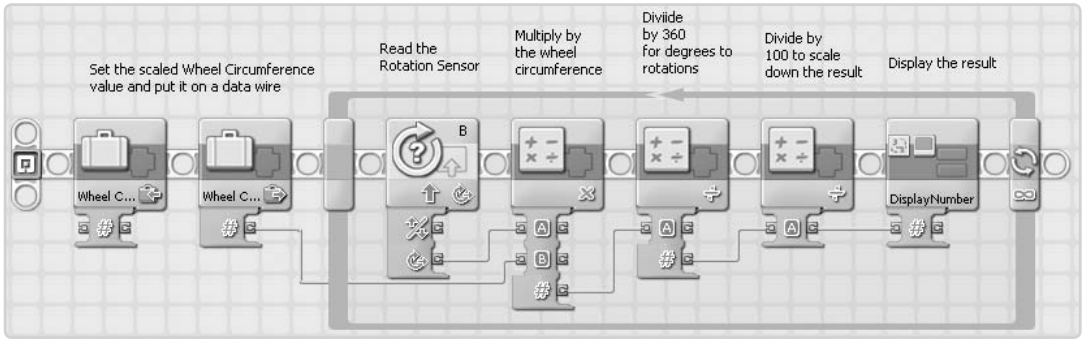


Figure 14-2: The Odometer program

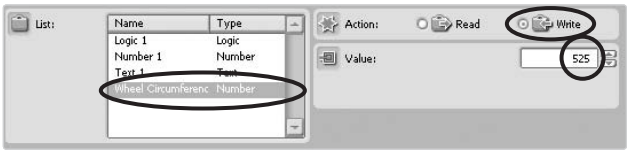


Figure 14-3: Setting the scaled Wheel Circumference value

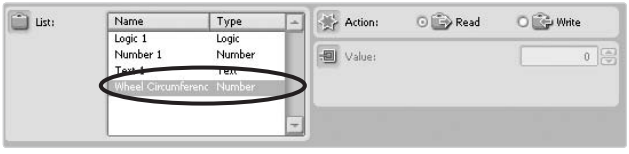


Figure 14-4: Reading the Wheel Circumference value

The program then enters a loop that reads the Rotation Sensor, converts the value from degrees to inches or centimeters, and displays the result. The Loop block is set to loop forever, as shown in Figure 14-5.

Figure 14-5: Looping forever

The Rotation Sensor block reads the distance the B motor has moved in degrees, as shown in Figure 14-6. The three Math blocks then multiply the value by the Wheel Circumference value, divide by 360, and then divide by 100. This is the NXT-G implementation of the following expression:

degrees × wheel circumference ÷ 360 ÷ 100

Figures 14-7 through 14-9 show the Configuration Panels for the Math blocks.

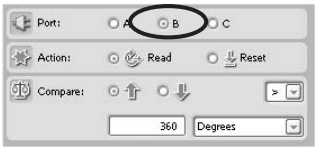


Figure 14-6: Reading the Rotation Sensor for the B motor

Figure 14-7: Multiplying the Rotation Sensor reading by the wheel circumference

Figure 14-8: Dividing by 360 to convert degrees to rotations

Figure 14-9: Dividing by 100 to scale down the result

Finally, the DisplayNumber block you created in Chapter 13 displays the value with a label and unit.

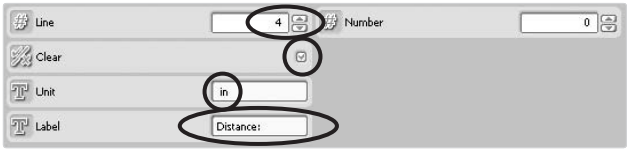


Figure 14-10: Displaying the value

When you run the program, it should display how far the B motor has moved. You could use this code as part of a larger program, for example to track how far the robot travels using the BumperBot or WallFollower program.

floating-point math

Because NXT-G 2.0 uses floating-point math, it avoids the problems associated with integer math described in the previous section. For many programs, this makes using numbers much easier; you just work with them the same way you would if you were using a calculator. However, floating-point numbers do have some limitations, and their effects are not always easy to understand.

range

The range of values that you can use with floating point numbers is quite large. You can have positive or negative values larger than 1×10^{30}, the number 1 followed by 30 zeros. You can also have values smaller than 1×10^{-31}, which has 30 zeros to the right of the decimal point followed by a 1. So, the range of values you can use is unlikely to cause you any problems.

precision

Floating-point math will almost always work just the way you expect. In the few cases when it doesn't, the problem will usually be because of the *precision* of floating-point numbers; that is, the number of digits you can expect to be correct. Although the range of floating-point numbers allows you to use as many as 30 digits, only the first 7 digits are guaranteed to be correct.

Table 14-3 shows some example values, the part of the value you can rely on, and the maximum error the value could have. For example, if the Math block gives you `123456789` as a result, you can be sure that the exact result is somewhere between 123,456,700 and 123,456,799.

table 14-3: floating-point numbers

number	part guaranteed to be correct	maximum error
123456789	123456700	89
123.456789	123.4567	0.000089
0.0123456789	0.01234567	0.0000000089

Notice that you can always rely on seven digits, regardless of where the decimal point is. This is where the "floating" part of floating-point math comes from. For most purposes, seven digits of precision is more than enough.

NOTE **Why isn't the number rounded to seven digits if that's all that are correct? People have 10 fingers, so we count by ten and find it convenient to think in terms of tens, hundreds, or thousands. Computers don't have fingers; they use electronic circuits that count by 2. Floating-point numbers actually are rounded, but they're rounded to powers of 2, which isn't very intuitive for humans.**

the number to text block

You've used the Number to Text block often to convert a number to text in order to display the value. When used with floating-point numbers, this block always rounds the value to two decimal places.

Figure 14-11 shows a very simple program for testing the Number to Text block. This program simply converts the number supplied by the Math block to text and displays the value.

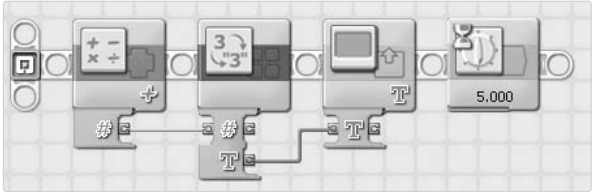


Figure 14-11: A simple program to test the Number to Text block

Table 14-4 shows the results from this program using some example values. As you can see, the displayed value will show at most two places to the right of the decimal point. Although this won't usually cause a problem, two different values could be displayed the same way, which can be a bit confusing. For example, rounding numbers to two decimal places gives the same result for the last two entries in Table 14-4: 0 and 0.0002 will both be displayed as 0.

table 14-4: number to text block results

math block value	displayed value
123	123
123.4	123.4
123.45	123.45
123.456	123.46
123.4567	123.46
0	0
0.0002	0

the random block

Now that you know how numbers work in NXT-G, you can move on to some of the other math-related blocks, starting with the Random block. This block is found in the Data group on the Complete Palette, as shown in Figure 14-12. Figure 14-13 shows how the block looks in a program.

Figure 14-12: The Random block on the Complete Palette

Figure 14-13: The Random block

A die is used for the picture on the Random block because in many games a die (or two or more dice) is used to generate a random number. Rolling a standard die will give you a random number between one and six. Each time you roll the die, you know you'll get a number in that range, but you don't know which one (which is why it's called a *random* number).

Like a die, the Random block is used to generate a random number. You can use this block to create robotic games or to add some randomness to your robot's behavior. Often a robot that is a little unpredictable can be more interesting or seem to have more personality.

You can set the range of values the Random block can generate using the Configuration Panel (shown in Figure 14-14). The default range is from 0 to 100, giving an output value that can be as small as 0 or as large as 100. You can change the range to suit your program; for example, to create a virtual die, you would set the Minimum value to 1 and the Maximum value to 6.

Figure 14-14: The Random block's Configuration Panel

You can set the Minimum and Maximum values using the two-sided slider or by typing the values in the boxes above the slider. Using the slider, you can set the values between 0 and 100. The Random block can work with values up to 32,767; you just need to enter the value in the box if it's greater than 100. The Minimum setting must be at least 0, which means that the block can't generate a negative number. If you need a negative random number, you can combine the Random block with a Math block. For example, the steering value for the Move block takes a value between -100 and 100. To generate a random steering value, you could have a Random block generate a value between 0 and 200 and then use a Math block to subtract 100 from the random number. This will give you a result between -100 and 100.

adding a random turn to BumperBot

In this section, you'll make a small change to the BumperBot program to make it a little more interesting. Recall that when the TriBot bumps into something, it backs up and turns in a different direction. The distance the robot turns doesn't need to be any particular value; you simply want to have the robot point in a different direction. You can use a Random block to control the distance the robot turns, which will make the program less predictable.

Figure 14-15 shows the part of the BumperBot program that you need to change. This is the code that runs after the Touch Sensor determines that the TriBot has run into something. The first five blocks make the robot back up, and the final Move block turns the robot.

The Duration setting for the Move block is set in the Configuration Panel to 350 degrees, as shown in Figure 14-16. To make the turn less predicable, add a Random block before the Move block, and connect the output data plug from the Random block to the Move block's Duration data plug, as shown in Figure 14-17.

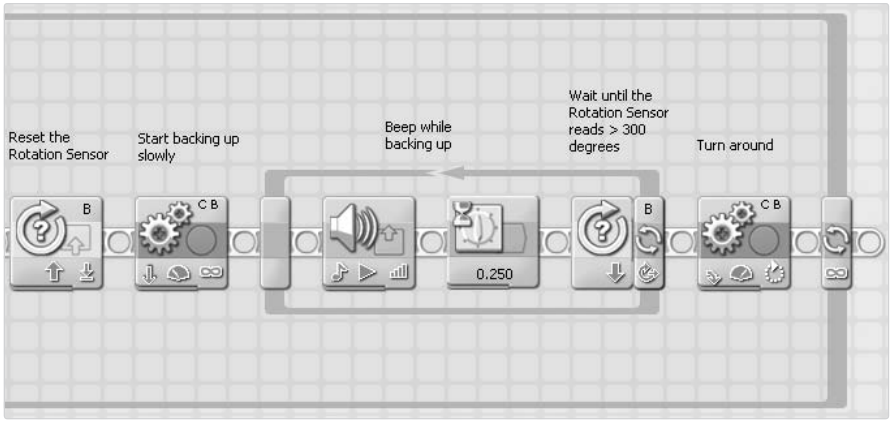


Figure 14-15: Backing up and turning around

Figure 14-16: The Configuration Panel for the Move block that turns the TriBot

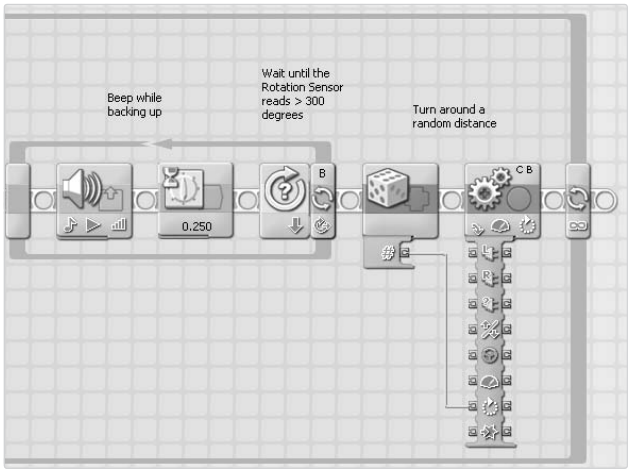


Figure 14-17: Connecting the Random block to the Move block

Now you need to set the range of values to use. The original program used 350 degrees, which turns the robot a little more than a quarter turn. I'll use 300 degrees for the Minimum and 2000 for the Maximum. With these values, the robot sometimes turns quickly and starts off again and sometimes spins in place for a while before resuming its journey around the room. Figure 14-18 shows the Configuration Panel for the Random block.

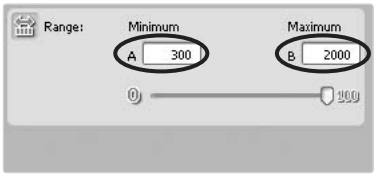


Figure 14-18: The Configuration Panel for the Random block

Run the program with these changes, and the TriBot should vary the amount it turns after bumping into something.

NOTE Notice that you didn't need to make any changes to the Move block's Configuration Panel. Attaching the data wire to the Duration input data plug causes the value set in the Configuration Panel to be ignored. When you're looking at a Configuration Panel to find out what a block does, make sure to also check the data wires attached to the block so you'll know which settings to ignore.

the logic block

Many of the programs presented so far make decisions using the program flow blocks, the sensor blocks, and the Compare block. These decisions involve a single condition, usually comparing the value from a sensor to a target value, with the result (either true or false) used in a Switch or Loop block. To put it another way, the programs are asking simple questions like "Is the Touch Sensor pressed?" or "Is the reading from the Light Sensor less than 50?"

The Logic block lets you combine multiple conditions, allowing your program to make more complex decisions. This

lets your program ask questions like "Is the Touch Sensor pressed and the Light Sensor reading greater than 50?"

You can find the Logic block with the other math-related blocks in the Data group on the Complete Palette, as shown in Figure 14-19. Figure 14-20 shows how the block looks when you add it to your program.

Figure 14-19: The Logic block on the Complete Palette

Figure 14-20: The Logic block

In many ways the Logic block is similar to the Math block, except that the Logic block works with logical values instead of numbers. The Logic block works the same way as the Math block; you select the operation you want to perform and supply the input values, using either data wires or the Configuration Panel (shown in Figure 14-21). Buttons are used to set the values in the Configuration Panel, with the check mark meaning true and the X meaning false.

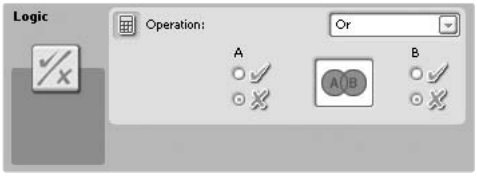


Figure 14-21: The Configuration Panel for the Logic block

The Logic block supports four operations: And, Or, Xor, and Not. The key to using this block successfully is understanding what each operations does.

- **And:** The result of the And operation will be true only if both input values are true. If either input value is false, then the result will be false.
- **Or:** The result of the Or operation will be true if either input value is true or if both input values are true. The result will be false only if both input values are false.
- **Xor:** Xor is an abbreviation for *Exclusive Or*. This is similar to the Or operation except that the result is false if both input values are true. This is the way the word *or* is often used in English; if your mother tells you that you can have ice cream or candy, she probably doesn't mean you can have both; rather, she expects you to pick one or the other.
- **Not:** This operation uses only the A input value and generates the opposite value. If the input value is true, then the output value will be false, and if the input value is false, then the output value will be true.

Logical operations are often described using a table that lists all the possible input values and the result for each operation. This is called a *truth table* because the values in the table are either true or false. Table 14-5 shows a truth table for the four operations supported by the Logic block. (Note that the result of the Not operation depends only on the Input A value.)

table 14-5: truth table for the logic block

input A	input B	or	and	xor	not
False	False	False	False	False	True
False	True	True	False	True	True
True	False	True	False	True	False
True	True	True	True	False	False

adding some logic to BumperBot

In this section, you'll make a change to the BumperBot program using the Logic block. Recall that the program keeps the TriBot moving forward until it runs into something. What if you want to limit how long the robot moves forward, perhaps to keep it from wandering too far or just to keep from getting bored? You'll change the program so that the TriBot stops and turns around if it bumps into something *or* if it travels for more than 20 seconds.

Figure 14-22 shows the code that moves the TriBot forward. The Move block starts the TriBot moving, and it keeps going until the Loop block ends when the Touch Sensor is pressed.

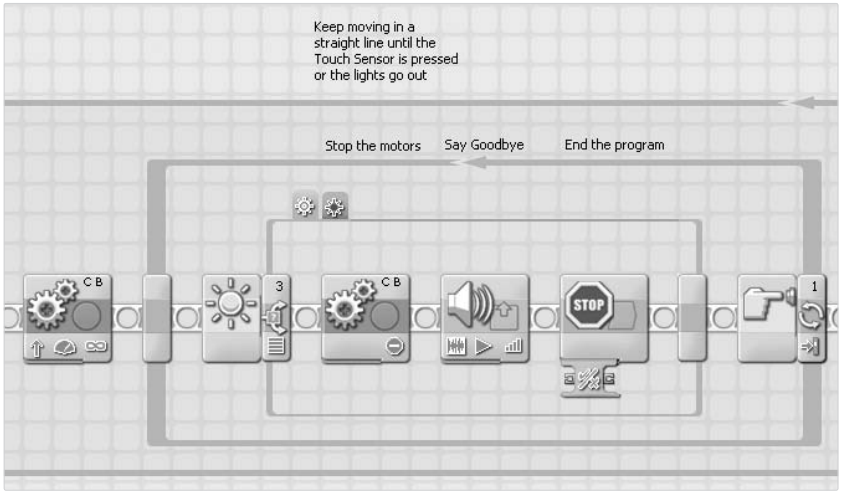


Figure 14-22: Moving forward until the Touch Sensor is pressed

You already know how to use the Touch Sensor to tell whether the TriBot has bumped into something. How can you tell whether it has traveled for more than 20 seconds? One way is to use a timer. You can use a Timer block to reset the timer before starting to move and then use another Timer block within the loop to tell when 20 seconds have passed.

The Loop block can be configured to check the Touch Sensor or the Timer, but it can't use both. To check both conditions, you need to change the way the loop is controlled. Instead of having the Loop block check the Touch Sensor, the two conditions can be checked using a Touch Sensor block and a Timer block. The output from these two blocks can then be combined using a Logic block, with the result used to exit the loop. I'll take you through these changes step-by-step.

1. Add a Timer block to the left of the Move block, and select **Reset** for the Action setting. Figure 14-23 shows the placement of the Timer block, and Figure 14-24 shows its Configuration Panel.

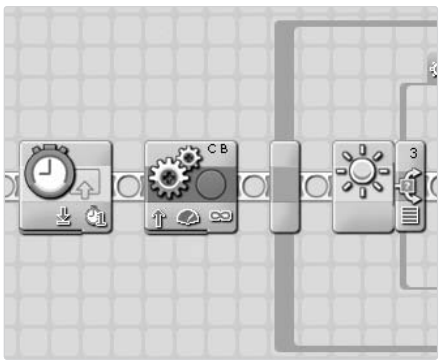
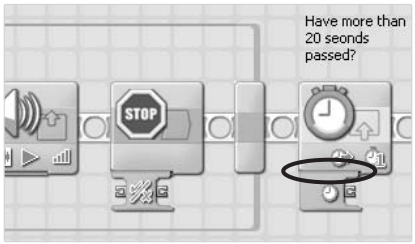


Figure 14-23: The placement of the Timer block

Figure 14-24: Resetting the timer

Add the two sensor blocks and the Logic block inside the Loop block, to the right of the Switch block.

2. Add a Timer block after the Switch block, and set the target value to **20**. The result of comparing the timer value to the target value is put on the Yes/No data plug. This plug is not shown when you first add the block to the program, so you'll need to open the complete data hub by clicking the tab at the bottom of the block, as shown here:



3. Add a Touch Sensor block after the Timer block. The default settings check for the sensor being pressed, so don't make any changes to this block.

4. Add a Logic block to the right of the Touch Sensor block. The default Operation is Or, and you'll be supplying the input values using data wires, so don't change any settings for this block.
5. Draw a data wire from the Touch Sensor block's Yes/No data plug to the Logic block's A data plug.
6. Draw a data wire from the Timer block's Yes/No data plug to the Logic block's B data plug.

Figure 14-25 shows this section of the program, and Figures 14-26 through 14-28 show the Configuration Panels for the three new blocks.

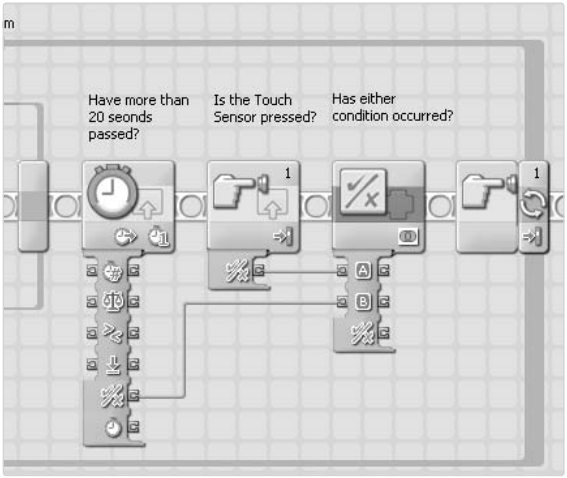


Figure 14-25: The program with the three new blocks added

Figure 14-26: Have more than 20 seconds passed?

Figure 14-27: Is the Touch Sensor pressed?

Figure 14-28: Is either condition true?

Finally, change the Loop block to use the output value from the Logic block to control the loop instead of using the Touch Sensor.

7. Select the Loop block, and change the Control setting from Sensor to **Logic**.
8. Draw a data wire from Logic block's Result data plug to the Loop block's Loop Condition data plug.

Figure 14-29 shows the program with these changes, and Figure 14-30 shows the Configuration Panel for the Loop block.

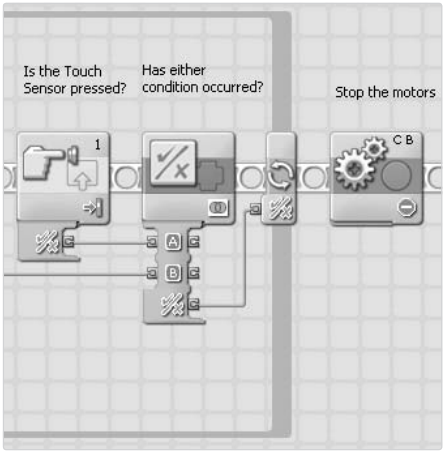


Figure 14-29: The Logic block connected to the Loop block

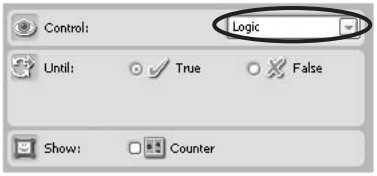


Figure 14-30: Exiting the loop when the Logic block output is true

Now when you run the program, the TriBot should go forward for a maximum of 20 seconds. If it doesn't bump into something within that time, it should turn and go off in a different direction.

the range block

The final math-related block is the Range block, which determines whether a number is inside or outside a range of numbers. For example, say you want to know whether the reading from the Light Sensor is between 40 and 60. You could pass the output value from a Light Sensor block to two Compare blocks and a Logic block, as shown in Figure 14-31.

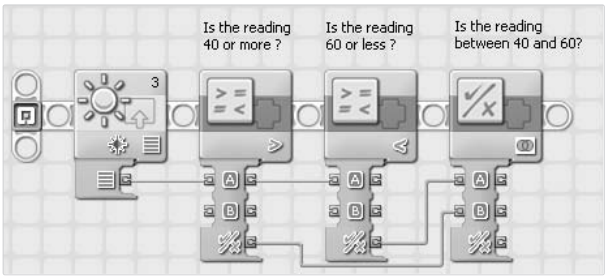


Figure 14-31: Is the Light Sensor reading between 40 and 60?

This approach will certainly work. However, checking to see whether a number is in a certain range is so useful that NXT-G provides a quicker way to do this. Using the Range block, you can perform the same comparison using one block instead of three. The Range block is on the Complete Palette in the Data group (shown in Figure 14-32). Figure 14-33 shows how the block will appear in your program.

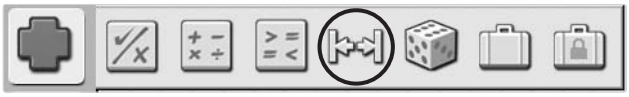


Figure 14-32: The Range block on the Complete Palette

Figure 14-33: The Range block

Figure 14-34 shows the Configuration Panel for the Range block. You can set the range by using the two-sided slider or by entering the values in the A and B boxes. Using the slider, you can select lower and upper limits between 0 and 100. To use a values greater than 100 or less than 0, enter the numbers in the boxes.

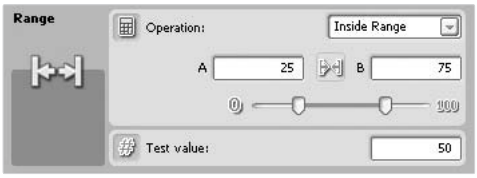


Figure 14-34: The Range block's Configuration Panel

The test value can be entered into the box on the Configuration Panel, although usually you'll supply the test value using a data wire.

There are two questions you can ask using the Range block: "Is the test value inside the range (between the lower and upper limit)?" and "Is the test value outside the range (less than the lower limit or greater than the upper limit)?" The Operation setting determines how the test value is compared with the range defined by the lower and upper limits (see Figure 14-35).

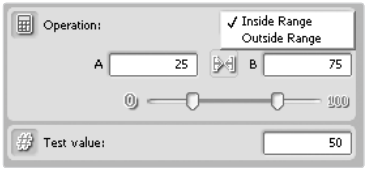


Figure 14-35: The Operation setting

improving RedOrBlue

You can use the Range block to improve the RedOrBlue program from Chapter 5. The original program uses the Light Sensor or the Color Sensor in Light Sensor mode and assumes that all objects are either red or blue. In this section, you'll change the program to eliminate this assumption, using the Range block to more accurately assign the color (red or blue) based on the Light Sensor reading.

The improvements made in this section can also be made to the RedOrBlueCount program. I'm using the simpler RedOrBlue program here to make it easier to concentrate on the use of the Range block. The original RedOrBlue program (shown in Figure 14-36) uses a Switch block to decide which color an object is based on the Light Sensor reading. One of the first steps in developing this program was determining the Light Sensor readings for red and blue objects. For red objects, I got a reading of 55, and for blue objects, I got a reading of 27. The Switch block compares the Light Sensor

reading with 42 (midway between 55 and 27) to classify objects as either red or blue.

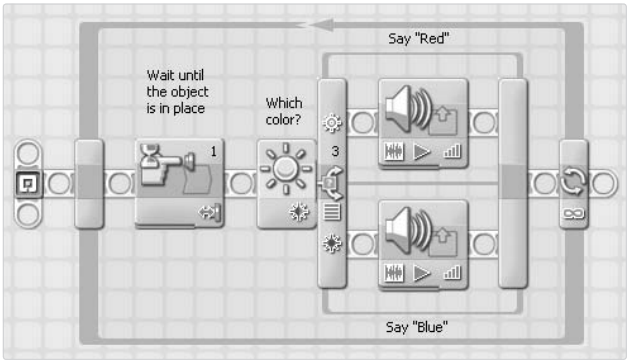


Figure 14-36: The original RedOrBlue program

Instead of identifying all objects that give a reading greater than 42 as red, you'll use the Range block to limit this to readings close to 55. Similarly, you'll use readings close to 27 for blue objects. For objects that give Light Sensor readings that aren't close to either 55 or 27, you'll have the robot say "Sorry" (the Sound Block doesn't have choices for "Unknown" or "I don't know").

To determine which ranges to use for each color, test several red and blue objects using the Feedback box on a Light Sensor block and record the values. Because the Light Sensor (and the Color Sensor using Light Sensor mode) measures only the amount of reflected light (not the color), you should use objects that are roughly the same shade of red or blue, or the range of values will be too large to be useful. The readings I got are between 51 and 61 for red objects and are between 24 and 32 for blue objects. I'll use these values for the Range blocks in the following programming instructions.

The first set of changes will use a Light Sensor block and a Range block to determine whether an object is red. You'll change the Switch block to use the output from the Range block instead of reading the Light Sensor, leaving the blocks inside the Switch block unchanged. The following instructions and figures use the Light Sensor block. If you're using the Color Sensor, then use the Color Sensor block, and set the Action setting to Light Sensor, as shown in Figure 14-37.

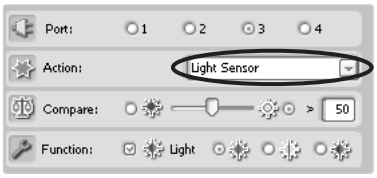
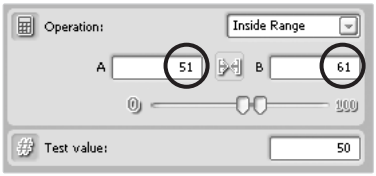


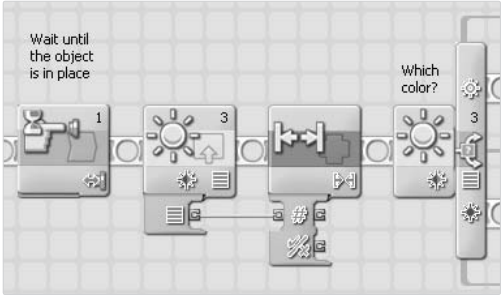
Figure 14-37: Using the Color Sensor in Light Sensor mode

Follow these steps to make the changes to the program:

1. Open the RedOrBlue program.
2. Add a Light Sensor block to the left of the Switch block. Don't make any changes to the Configuration Panel.
3. Add a Range block to the right of the Light Sensor block. Set the A and B values to **51** and **61**. The Configuration Panel should look like this:

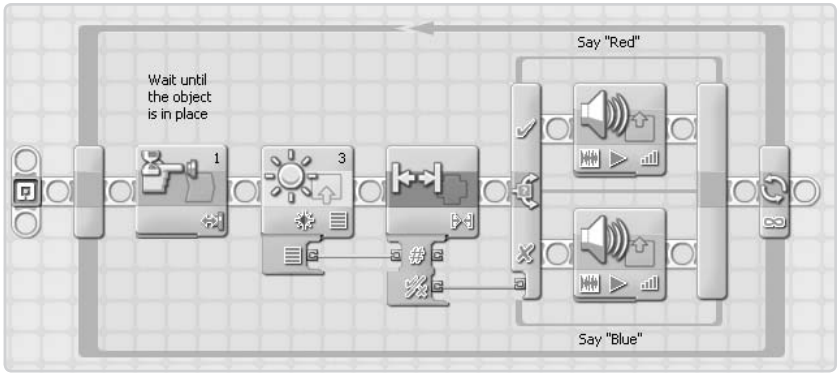


4. Draw a data wire from the Light Sensor block's Intensity data plug to the Range block's Test Value data plug. This section of the program should look like this:



5. Select the Switch block, and change the Control from Sensor to **Value**. The Configuration Panel should look like this:

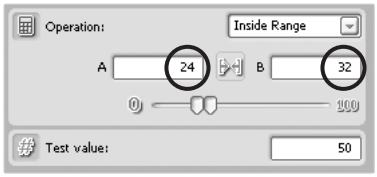
6. Draw a data wire from the Range block's Yes/No data plug to the Switch block's Value data plug. The program should now look like this:



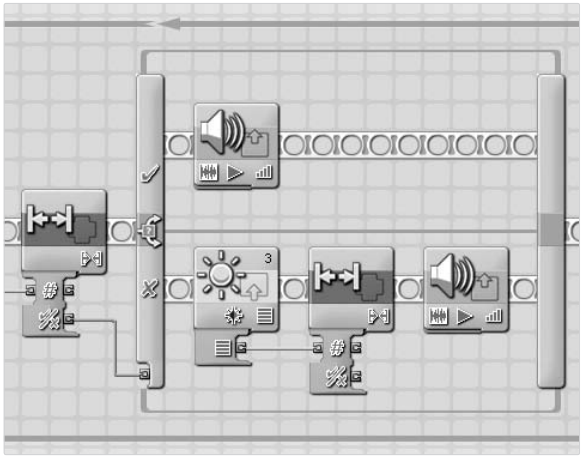
When you test your program, it should correctly identify all the red objects. You may need to adjust the values set in the Range block slightly if your lighting conditions and selection of test objects is different from mine. At this point, all nonred objects are identified as blue. You'll address this next.

To identify blue objects, you'll again use a Light Sensor block followed by Range block. You'll put these on the lower Sequence Beam of the Switch block, which is used only for nonred objects and then add another Switch block to say either "Blue" or "Sorry" as appropriate. Follow these steps to make the changes:

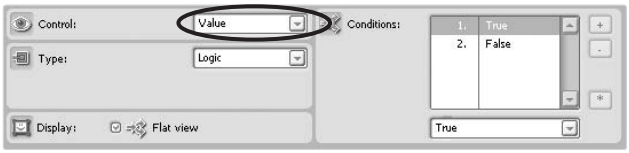
7. Add a Light Sensor block to the lower Sequence Beam of the Switch Block, to the left of the Sound block. Don't make any changes to the Configuration Panel.
8. Add a Range block to the right of the Light Sensor block, and set the A and B values to **24** and **32**. The Configuration Panel should look like this:



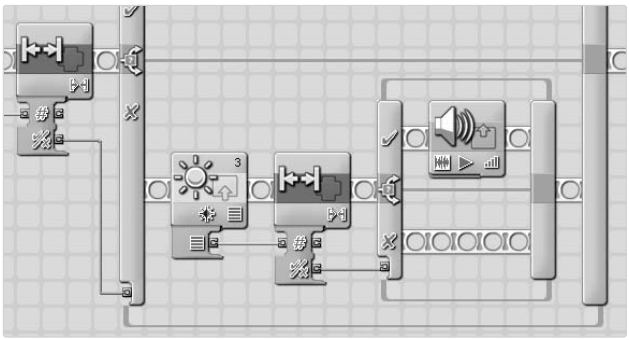
9. Draw a data wire from the Light Sensor block's Intensity data plug to the Range block's Test Value data plug. This section of the program should look like this:



10. Add a Switch block between the Range block and the Sound block. Set the Control to **Value**, and draw a data wire from the Range block's Yes/No data plug to the Switch block's Value data plug. The Configuration Panel should look like this:

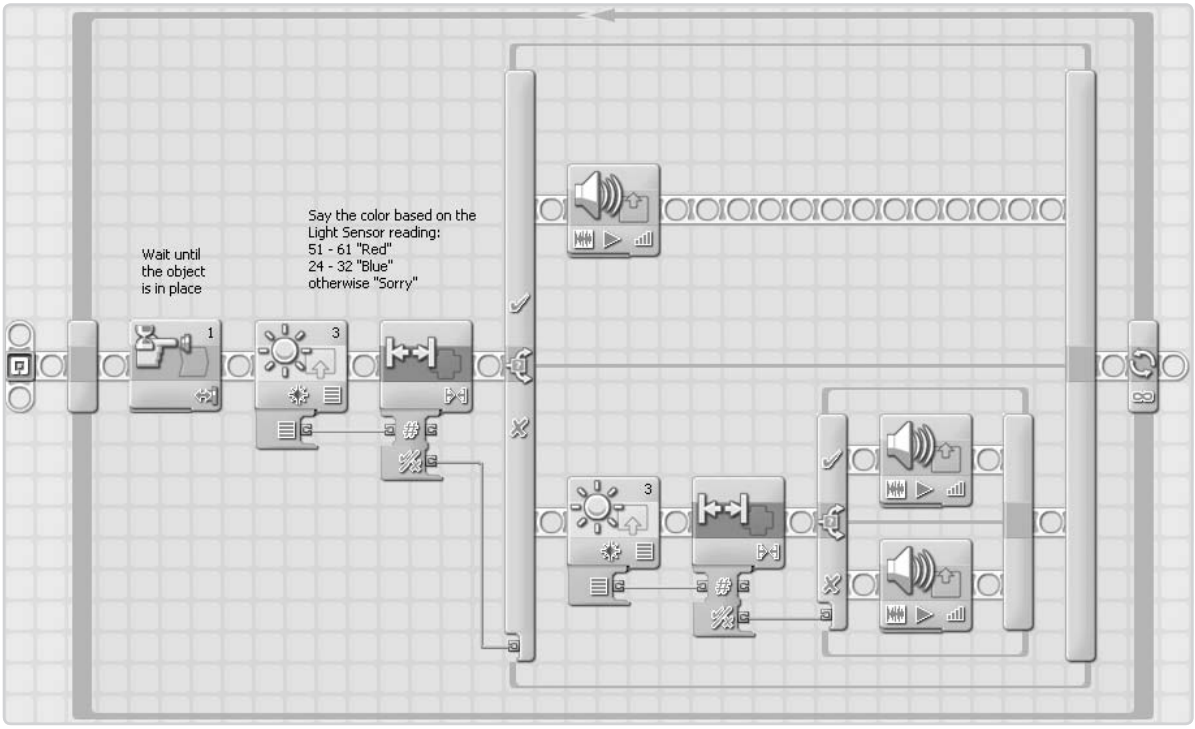


11. Drag the existing Sound block (the one that says *Blue*) onto the upper Sequence Beam of the new Switch block. This part of the program should look like this:



12. Add a Sound block to the lower Sequence Beam of the new Switch block. Select **Sorry** from the File list. The Configuration Panel should look like this:

The completed program should look like this:



Now the program should correctly identify red or blue objects and say "Sorry" for all others. Again, you may need to adjust the values for the second Range block to make the program work for all blue objects.

improving RedOrBlueColorMode

The RedOrBlueColorMode program from Chapter 5 (shown in Figure 14-38) is similar to the RedOrBlue program, except that it uses the Color Sensor to determine the color directly. The program does a good job of identifying red objects, but it calls all other objects blue. You could fix this problem using nested Switch blocks like you did for the RedOrBlue program, but there is an easier way.

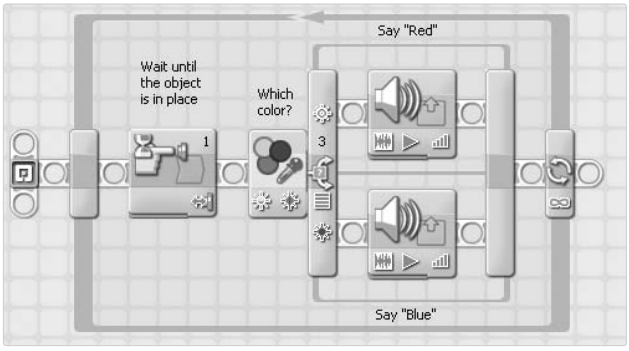


Figure 14-38: The RedOrBlueColorMode program

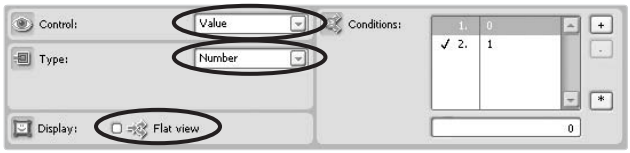
The Color Sensor block can determine the color of an object and generate an output value using the values in Table 14-6. In the previous section, the Switch blocks made decisions based on a logical value passed from a Range block. But since a Switch block can also use a number to make a decision, we can improve the program by changing the Switch block to use the output from a Color Sensor block. The Switch block will need three choices: one for red, one for blue, and one for all other colors.

table 14-6: color sensor output values

number	color
1	Black
2	Blue
3	Green
4	Yellow
5	Red
6	White

Follow these steps to make the changes:

1. Open the RedOrBlueColorMode program.
2. Add a Color Sensor block between the Touch Sensor block and the Switch block. Don't make any changes to the Configuration Panel.
3. Select the Switch block. Change Control to **Value** and Type to **Number**.
4. Uncheck the **Flat view** option so that you can use three conditions. The Configuration Panel should look like this:

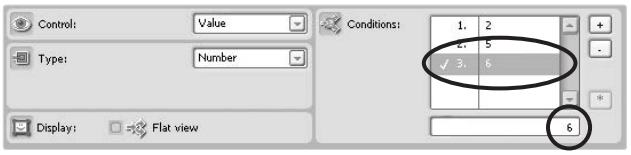


Now check the Configuration Panels of the two Sound blocks to see which tab they are on. The Sound block that says "Blue" should be on the first tab and will be used when the input is 0. The Sound block that says "Red" should be on the second tab and will be used when the input is 1 and, because it's selected as the default, will also be used for any value other than 0. Follow these steps to change the values to match the output from the Color Sensor block and to add the third choice to be used for objects that are neither red nor blue:

5. Select the top item in the Conditions list. In the box at the bottom of the panel, change the 0 to **2** to match the Color Sensor output for a blue object. (Once you change the value, the Conditions list will be reordered.) The Configuration Panel should look like this:

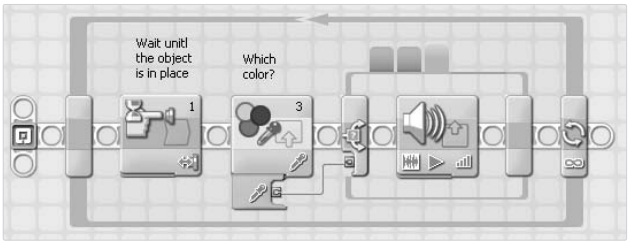
6. Select the top item in the Conditions list, which should now match 1. In the box at the bottom of the panel, change the 1 to **5** to match the Color Sensor output for a red object. The Configuration Panel should now look like this:

7. Click the **+** button to add a new condition to the list. The value for the new item will be 6 because the largest value in the list is 5. Click the ***** button to make this item the default. We don't really care what the value is; we just need this condition to match any value other than 2 or 5. The Configuration Panel should now look like this:



8. Select the third tab of the Switch block if it isn't already selected. Place a Sound block on the Sequence Beam, and select **Sorry** from the File list. The Configuration Panel should look like this:

9. Draw a data wire from the Color Sensor block's Detected Color data plug to the Switch block's Value data plug. The program should now look like this:



When you run this version of the program, you should notice that it does a much better job of identifying a wider range of shades of blue and red, in addition to being much simpler than the program that uses Light Sensor mode. You can also easily extend the program to identify the other colors by adding a few more tabs to the Switch block.

conclusion

This chapter has covered using numbers and logic in your programs. The Math block is used for operating on numbers, and in most cases will behave exactly as you'd expect. Knowing how integer or floating-point math works (depending on which version of NXT-G you're using) will help you avoid the most common problems encountered with computer math.

The Logic block lets you write programs that make complex decisions, such as combining the input from multiple sensors. The Range block gives you a convenient way to perform the common operation of testing a value to see whether it's in a certain range. The other block introduced in this chapter was the Random block, which you can use to add a little unpredictability to your programs and personality to your robots.

15

files

A *file* is a collection of information stored on a computer (which in our case is the NXT). In this chapter, you'll learn how to use the File Access block to create and use files in your programs. You'll make some changes to the RedOrBlueCount program to save the number of objects in a data file and then restore the values the next time the program starts. You'll also learn about managing your NXT's memory, including how to delete files or transfer them between the NXT and your computer.

using files

Your computer uses files to store music, pictures, programs, word processing documents, and numerous other kinds of information. The NXT also uses files to store many different types of information, such as your programs, the images used by the Display block, and the sounds used by the Sound block.

The File Access block allows you to create your own files on the NXT, which you can use to store any data that your programs use. The information you store in a file is *persistent*, meaning that it's still available after your program ends, even if you turn off the NXT. Persistence lets you store information from a program and use it later in the same or a different program. Some common uses for files in NXT-G programs are as follows:

* Storing data collected by the program as it runs. For example, in "Saving the RedOrBlueCount Data" on page 197, you'll use a file to store the number of objects counted by the RedOrBlueCount program. Other examples include high scores for a game or a map generated by a maze-solving robot.
* Storing program settings such as the speed the robot should use or trigger values for sensors. Many programs need to be adjusted for a particular environment, and storing the program settings in a file makes it possible for you to easily customize the program.
* *Data logging* or collecting sensor data as part of an experiment or a test program. The versatility and ease of use of the NXT kit make it an ideal data logging tool for classroom science experiments. Collecting and analyzing the data from small test programs can teach you a lot about how the sensors work. I'll discuss data logging in depth in Chapter 16.

the file access block

Working with files is the job of the File Access block, found in the Advanced group on the Complete Palette (shown in Figure 15-1). Figure 15-2 shows how this block should look when you add it to your program.

Figure 15-1: The File Access block on the Complete Palette

Figure 15-2: The File Access block

Figure 15-3: The File Access block's Configuration Panel

the filename

All NXT files have a name. The first item you usually set in the Configuration Panel (Figure 15-3) is the name of the file to use. Filenames can be up to 15 characters long and can include numbers, letters, spaces, and most of the special characters such as * and #. Try to use meaningful filenames that will tell you something about the actual content of your files.

When the NXT is connected to the MINDSTORMS software with the USB cable or via a Bluetooth connection, the files already on the NXT will appear in the File list, as shown in Figure 15-4. To reuse a file that's already on the NXT, select it from the list, and the Name box will be filled in automatically. This is both more convenient and less error prone than entering the name yourself. To create a new file, or if the NXT isn't connected to the MINDSTORMS software, enter the name in the Name box.

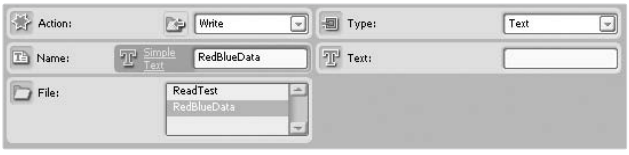


Figure 15-4: Selecting the name from the File list

the action setting

The Action setting tells the block what you want to do with the file. There are four choices: Write, Read, Delete, and Close.

- **Write**: Stores information in a file. The file will be created if it doesn't already exist. The File Access block always writes new data at the end of the file, so if the file already exists, the new data is added on at the end.
- **Read**: Retrieves information from a file. The value read from the file is passed to other blocks in the program using a data wire.
- **Delete**: Deletes a file. To replace the information in a file, first delete the file and then write the new value. For example, to set the high score of a game, first delete the file containing the old high score, and then write the new value.
- **Close**: Closes the file, which tells the NXT that you are through using it. After using the Write Action to add data to a file, you need to close the file before you can read from it or delete it. Similarly, after using the Read action, you must close the file before you can write to or delete the file.

the type setting

The File Access block can read and write both numbers and text values. Select the appropriate choice for the Type setting to tell the block which data type you are using.

When writing to the file, the value can be supplied via a data wire or the Configuration Panel. With Text selected for Type, the Configuration Panel will appear as shown in Figure 15-4, and you can enter the text to write to the file in the Text box. When writing a number, the Configuration Panel will appear as shown in Figure 15-5, and you can enter the number to write to the file in the Number box.

Figure 15-5: Configuration Panel settings for writing a number

When the Action is set to Read, the data will be available on either the Text out or Number out data plug (shown in Figure 15-6), depending on the Type setting.

Figure 15-6: The File Access block's Text out and Number out data plugs

saving the RedOrBlueCount data

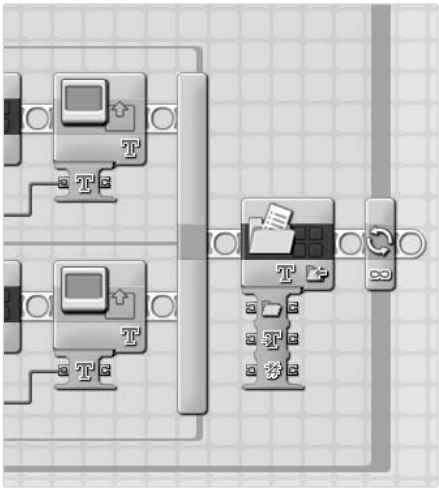
Recall that the RedOrBlueCount program presented in Chapter 11 identifies and counts red and blue objects. In this section, you'll change the program to save the number of red and blue objects to a file. Figure 15-7 shows the main part of the original program. The Switch block uses the Light (or Color) Sensor to determine the object's color; then the appropriate variable is updated, and the new total is displayed. You'll add code after the Switch block to save the values of the two variables, Total Red and Total Blue, to a file named *RedBlueCounts*.

Saving these two values is a four-step process:

1. Delete the existing file. If the file already exists (it will after the first time through the loop), you need to delete it before writing the new values, or those values will be added onto the end of the file instead of replacing the current values.
2. Write the Total Red value.
3. Write the Total Blue value.
4. Close the file, which tells NXT that you are finished writing to it. Close the file each time through the loop so that it can be deleted the next time around.

The following instructions will walk you through the entire process, starting with deleting the file if it already exists:

1. Open the RedOrBlueCount program.
2. Add a File Access block just after the Switch block. Be sure that it's inside the Loop block, as shown here:



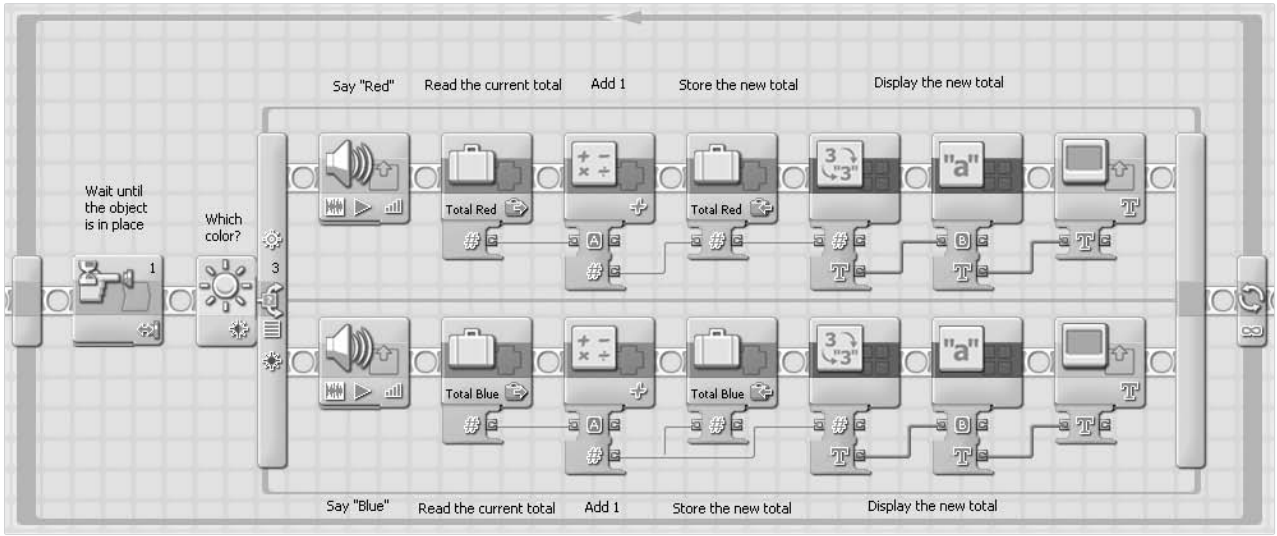
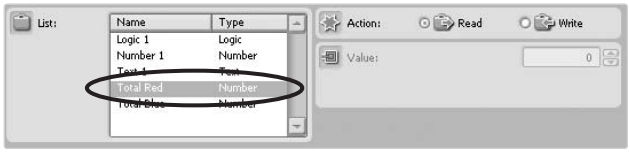


Figure 15-7: Counting red and blue objects

3. In the Configuration Panel, select **Delete** for the Action setting, and enter **RedBlueCounts** for the filename.

To write the number of red objects to the file, use a Variable block to get the current value.

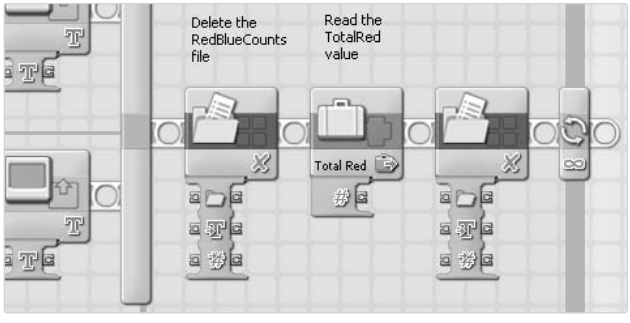
4. Add a Variable block after the File Access block, and select **Total Red** from the list of variables.



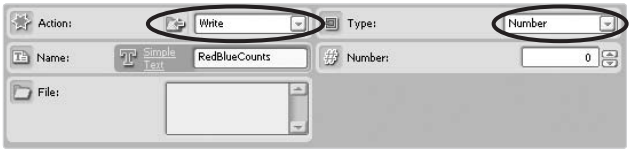
Next, you'll use three more File Access blocks to write the values to the file, all with the same name.

NOTE **The most common problem that I've encountered when using using files is misspelling the filename. If one of the four blocks has the filename misspelled, the program won't work correctly. To avoid this problem, copy the existing File Access block (the one used to delete the file) instead of adding a new one by holding down the CTRL key while clicking and dragging the block to the right of the Variable block. Move the mouse slowly and let the Loop block expand to make room for the new block before releasing the mouse button.**

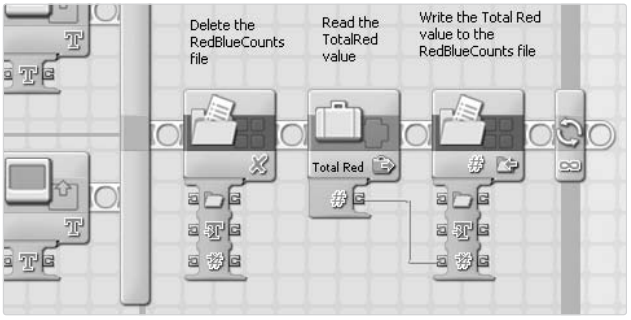
5. Copy the first File Access block, and place the copy after the Variable block.



6. Make sure the new block is selected, and then use the Configuration Panel to set the Action item to **Write**. Set the Type item to **Number** to match the data type of the Total Red variable.

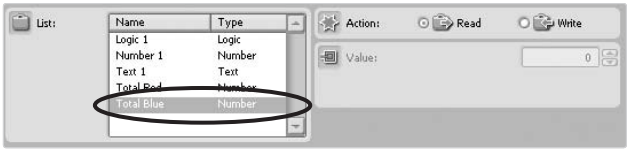


7. Draw a data wire from the Variable block's Value plug to the File Access block's Number plug. This section of the program now should look like this:



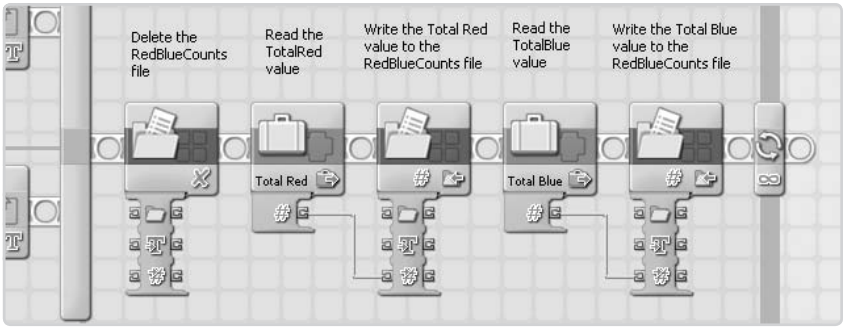
The new File Access block will first create the *RedBlueCounts* file and then write the value from the Total Red variable. Now you need to add Variable and File Access blocks to write the Total Blue value to the file.

8. Add a Variable block to the right of the File Access block, and select **Total Blue** from the list of variables.



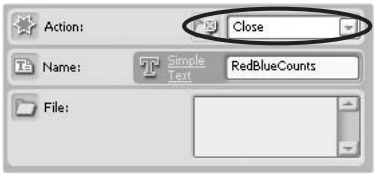
9. Copy the last File Access block we added (the one that writes to the file), and place the copy after the new Variable block. This block will write the Total Blue value to the file. None of the block's configuration settings needs to change since this block also writes a number to the *RedBlueCounts* file.

10. Draw a data wire from the Variable block's Value plug to the File Access block's Number plug. This section of the program now should look like this:

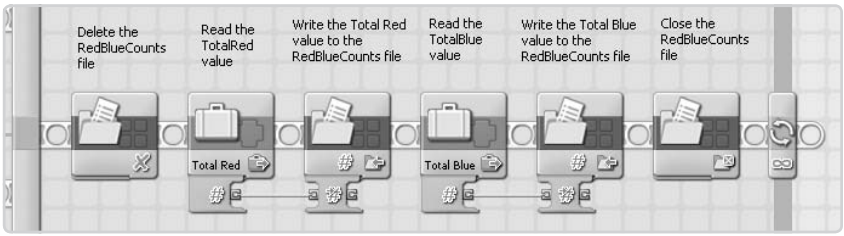


Finally, close the file. Once again you'll copy the previous File Access block and then change the Action setting to ensure that the same filename is used.

11. Copy the previous File Access block, and place the copy just to the right of the existing block. Change the Action setting to **Close**.



12. Close the data hubs on the blocks you've added. The completed section of code should look like this:



Now run the program and test it with several red and blue objects. The program should write the count of red and blue objects to a file, although there won't be any visible difference in the program's behavior.

You can tell that the file has been created by connecting the NXT to the MINDSTORMS software and selecting one of the File Access blocks. If the program worked as expected, then *RedBlueCounts* should appear in the Configuration Panel's File list, as shown in Figure 15-8.

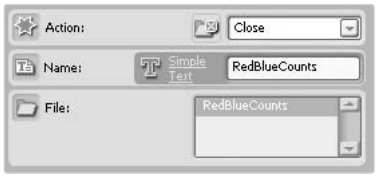


Figure 15-8: RedBlueCounts is in the File list.

NOTE **Actually, all this tells you is that the file was created, not that the correct information was written to the file. Later in this chapter you'll write the FileReader program to read and display the values in a file. Then you can then use this program to make sure the values are saved correctly.**

checking for errors

One interesting aspect of the File Access block is that it may not be able to perform the task you've given it. For example, the block can't read from a file that doesn't exist, and it can't write to a file if the NXT's memory is full. The block's Error data plug (shown in Figure 15-9) allows you to check for these kinds of situations. The data plug reports a logic value, namely, true if there was an error and false if the operation was successful.

Figure 15-9: The File Access block's Error data plug

Any File Access block can have an error. When designing your program, you need to decide whether you should add code to check for an error and what the program should do if a block fails.

I didn't include any error checking to the RedOrBlueCount program, even though there could be a problem. Let's look at what could go wrong with the File Access blocks and decide whether you should be checking for errors:

- **Deleting the file** can fail if the file doesn't exist. This isn't a problem because the reason for including this block is to make sure the file doesn't exist before you start writing to it. In fact, the first time you run the program, the file won't exist.
- **Writing the Total Red value** can fail if the file can't be created or the value can't be written because the NXT is out of memory.
- **Writing the Total Blue value** can fail if the value can't be written because the NXT is out of memory.
- **Closing the file** can fail if the file wasn't created by the block that writes the Total Red value. This block will fail only if the earlier block fails, so you can safely ignore this error.

You can ignore errors from the File Access blocks that delete and close the file, but what about the errors from the blocks that write the data? These blocks can fail if the NXT's memory is full when you run the program, and you can detect this situation by checking the Error data plug on the two File Access blocks that write to the file.

What should the program do if it can't write to the file? One possibility is to write a message on the screen, wait for the user to press a button to acknowledge the message, and then stop the program. Figure 15-10 shows how the code to check for an error when writing the Total Red value might look. Note that this code doesn't do anything to fix the problem; it just alerts the user so that the error isn't silently ignored.

NOTE **Adding error checking code both takes up space and can disrupt the visual flow of the program, making it more difficult to read and understand. For a simple test program, I wouldn't bother adding this code. On the other hand, if this program was for a class project, a demonstration, or some other important event where failure is a serious issue, then I would definitely add some error checking code to avoid any unpleasant surprises.**

the FileReader program

FileReader is a very simple program that displays the contents of a file on the NXT's screen, one value at a time. This program uses two properties of the File Access block:

- All values written to the file are text values. When you write a number, the File Access block converts it to a text value before writing it to the file. The block also converts a value from text to a number when reading the file, which means that you can read all the data from a file as text values, even if some or all of the data was originally numbers.
- When reading from a file, the Error value will be true when all the data has been read. To read all the values in a file, you can read the values in a loop that continues until the Error value is true.

Figure 15-11 shows the program. The File Access block reads a value from the file as text. If there is no error, the value is displayed, and the program waits for you to press the Enter button. The blocks within the Switch block are on the false tab, because the Error value will be false if the read is successful. The Loop block repeats until the File Access block's Error value is true.

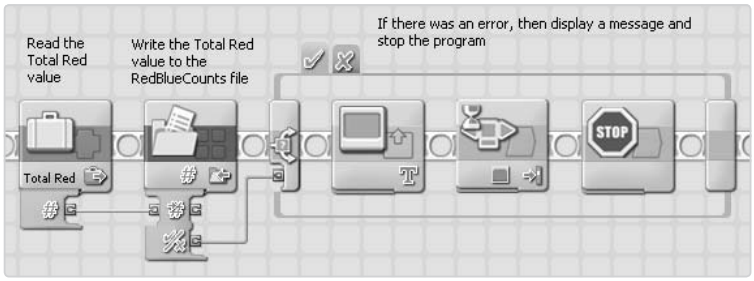


Figure 15-10: Error checking code

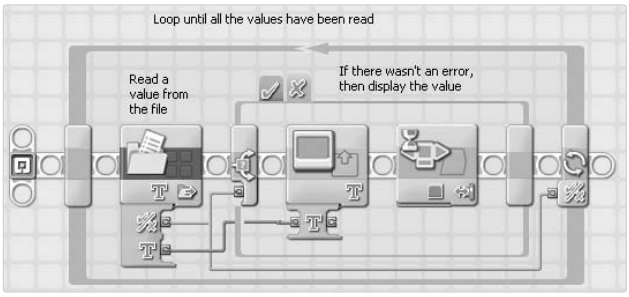


Figure 15-11: The FileReader program

Figures 15-12 through 15-16 show the Configuration Panel for each block. The File Access block is configured to use the *RedBlueCounts* file. Change this setting to view a different file.

Figure 15-12: Configuration Panel for the File Access block

The Switch block uses the File Access block's Error value. The Flat view option must be unchecked in order to connect the data wire from the File Access block to the Display block.

Figure 15-13: Configuration Panel for the Switch block

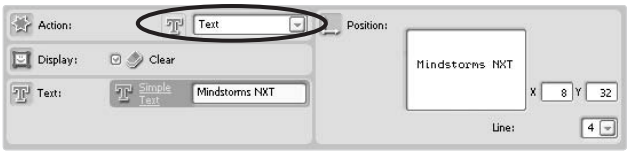


Figure 15-14: Configuration Panel for the Display block

The Wait block pauses the program until you press and release the NXT's Enter button to give you time to read the value.

Figure 15-15: Configuration Panel for the File Access block

The Loop block continues until the File Access block's Error value is true, which will happen after all the data has been read from the file.

Figure 15-16: Configuration Panel for the File Access block

When you run the program, you should see the totals from your last test run of the RedOrBlueCount program. The Total Red value should be displayed first, followed by the Total Blue value. The program should stop after displaying these two values.

restoring the RedOrBlueCount data

In this section, you'll change the RedOrBlueCount program so that the Total Red and Total Blue variables are initialized from the values saved in the *RedBlueCounts* file, instead of starting at zero. For each variable, you'll use a File Access block to read the saved value from the file. If the file doesn't already exist, the File Access block will fail, and the value from the Error data plug will be true. If this happens, you'll just set the variable to zero.

The code that writes the values to the file writes Total Red first and then Total Blue. The code that reads the values from the file needs to use the same order, so you'll initialize Total Red and then Total Blue. After reading the two values from the file, you need to close the file, or you won't be able to delete it later in the program. Here's the pseudocode for initializing the two variables:

```
read a number from the RedBlueCounts file
store the number in the Total Red variable
if there was an error reading the file then
   set Total Red to 0
end if
read a number from the RedBlueCounts file
store the number in the Total Blue variable
if there was an error reading the file then
   set Total Blue to 0
end if
close the RedBlueCounts file
```

For the following programming instructions, I've turned on the NXT and connected it to the MINDSTORMS software to make the *RedBlueCounts* file appear in the File Access block's list of files. This makes writing the program a little easier. Figure 15-17 shows the two Variable blocks that perform the initialization in the original program. Instead of deleting these blocks, you'll add the code to read the values from the file and use these blocks if there is an error.

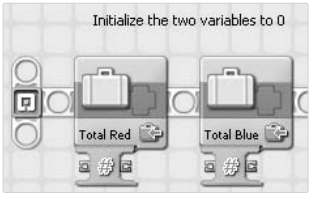
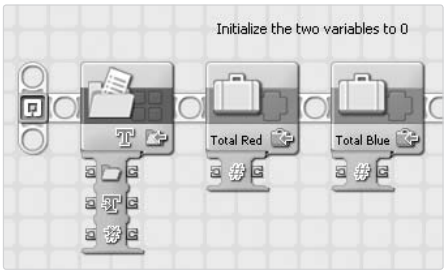


Figure 15-17: Original initialization code

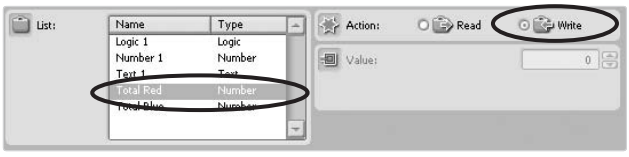
Follow these steps to make the necessary changes to the program:

1. Open the RedOrBlueCount program (if it's not already open).
2. Drag a File Access block to the beginning of the program.



3. Select **RedBlueCounts** from the File list. If you don't have the NXT connected to the MINDSTORMS software, you'll need to enter the filename in the Name box.
4. Set the Action setting to **Read** and the Type setting to **Number**.

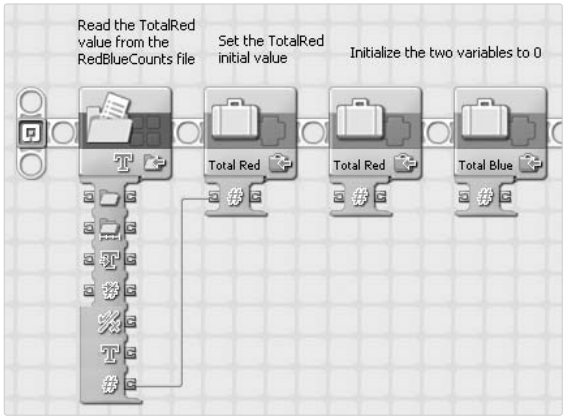
5. Add a Variable block to the right of the File Access block, and select **Total Red** from the list of variables. Set the Action setting to **Write**.



To store the value read from the file in the Total Red variable, you need to draw a data wire from the File Access block to the Variable block. When you first add a File Access block to the program, only three data plugs are displayed, and the one you need to use, Number out, is hidden. Before connecting the data wire, you must open the full data hub.

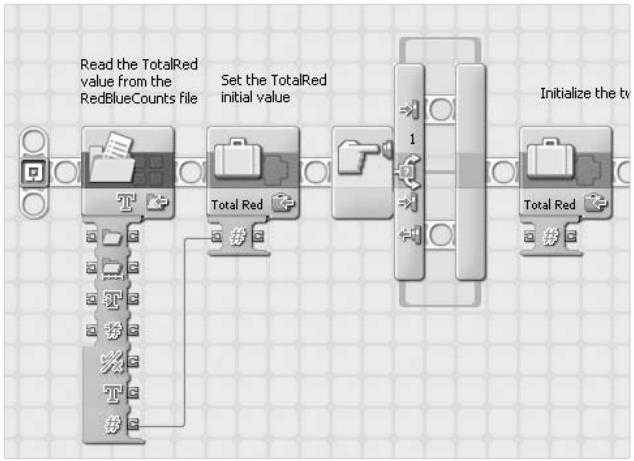
6. Click the tab at the bottom of the File Access block to open the full data hub.

7. Draw a data wire from the File Access block's Number out data plug to the Variable block's Value data plug.



Now you'll add the code to check for an error. You'll add a Switch block that uses the value from the File Access block's Error data plug and then move the Variable block that sets Total Red to 0 into the Switch block. If there is an error, the variable will be correctly initialized to 0, and if there isn't an error, the Switch block will do nothing.

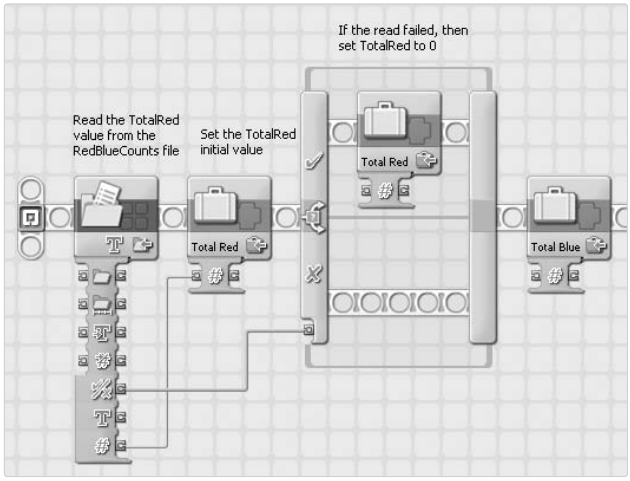
8. Add a Switch block after the Variable block that uses the value from the File Access block.



9. Set the Switch block's Control item to **Value**.

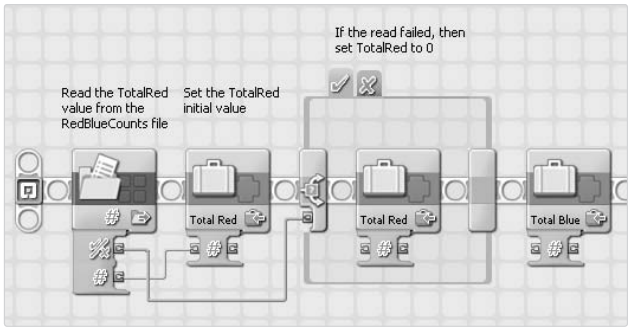
10. Draw a data wire from the File Access block's Error data plug to the Switch block's data plug.

11. Drag the Variable block that is to the right of the Switch block onto the Switch block's upper Sequence Beam. This section of the program should now look like this:



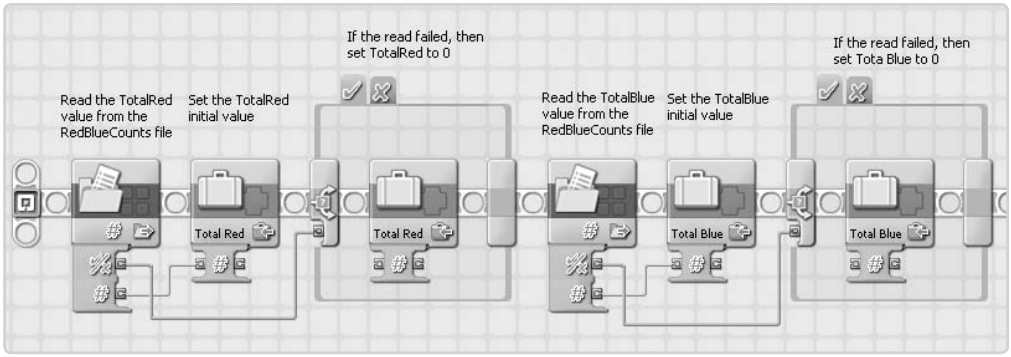
NOTE **You could have put both Variable blocks within the Switch block so that the value from the File Access block is written to the variable only when the read operation is successful. I prefer to include only the error handling code within the Switch block because that provides a better separation from the normal program flow, making the program easier to understand.**

12. Close the File Access block's data hub, and unselect the Switch block's Flat view. This section of the program should now look like this:



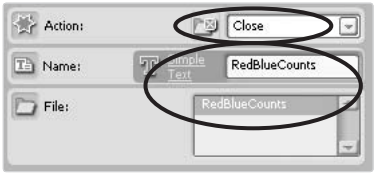
Now you need to make similar changes to initialize the Total Blue variable. You should be able to accomplish this task without detailed step-by-step instructions. The settings of the File Access and Switch blocks should be identical to the ones we used for initializing the Total Red value. The Variable blocks should use the Total Blue variable instead of

Total Red. When you have completed the changes, the beginning of the program should look like this:

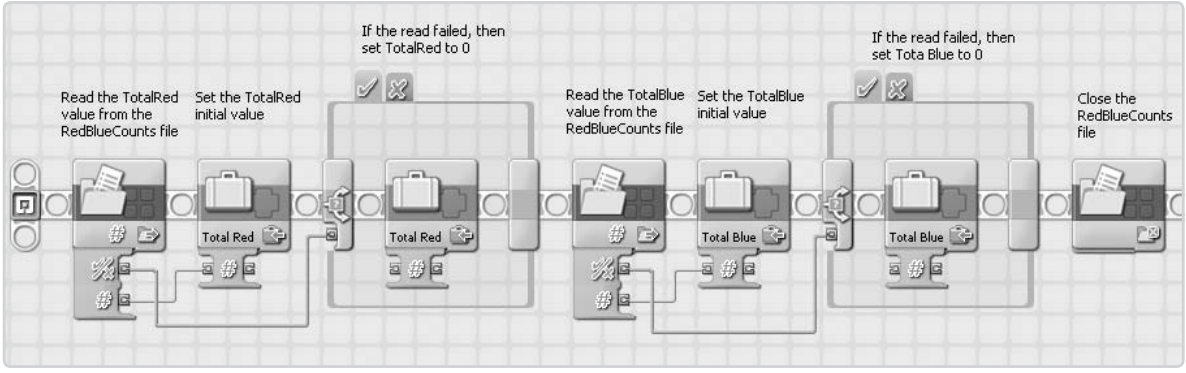


Once both values have been read from the file, the program should close the file.

13. Add a File Access block at the end of the new code.
14. Select **RedBlueCounts** from the File list, and set the Action item to **Close**.



The completed code for initializing the two variables should look like this:



You still need to make one more set of changes before testing the program. After the code that initializes the variables and before the start of the main loop are two Display blocks that print the starting values of the two variables, as shown in Figure 15-18. These two blocks print the text values `Red: 0` and `Blue: 0`, which worked fine when the counters always started at zero. Now that the values might not be zero, you need to add some code to print the correct values. To display each value, you'll use a Variable block to read the value and a DisplayNumber block to add the label and display the value.

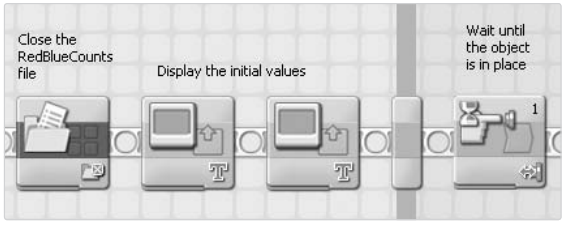


Figure 15-18: Original code to display the initial values

Figure 15-19 shows the changes needed to print the correct count of red objects. The Variable block (shown in Figure 15-20) reads the current Total Red value. The DisplayNumber block (shown in Figure 15-21) adds the label "`Red: `" before the value and displays the result after clearing the display. Remember to put a space after the colon so that the number doesn't get squished up against the label.

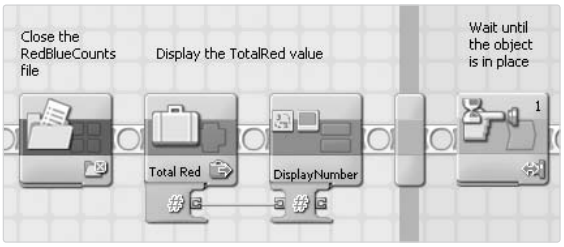


Figure 15-19: Displaying the Total Red value

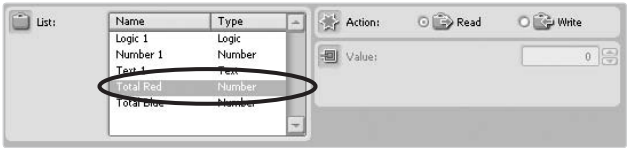


Figure 15-20: Reading the current Total Red value

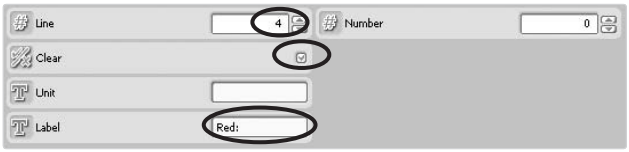


Figure 15-21: Adding a label and displaying the value

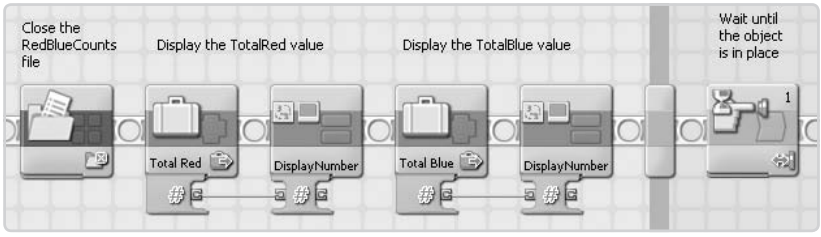


Figure 15-22: Displaying the Total Blue value

The code to display the number of blue objects (shown in Figure 15-22) is similar. The Variable block reads the Total Blue variable (shown in Figure 15-23), and the Display-Number block (shown in Figure 15-24) displays the value with the label "`Blue: `" on line 5 without first clearing the screen.

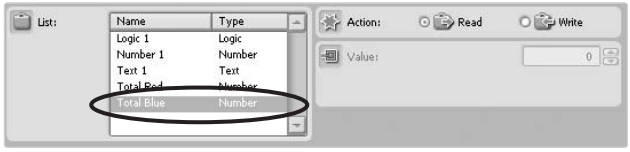


Figure 15-23: Reading the current Total Blue value

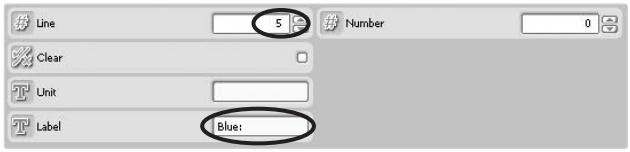


Figure 15-24: Adding a label and displaying the value

When you run the program, it should start counting using the previous values, instead of starting at zero. Each time the program runs, it will add on to the previous totals. To restart the counters (set them back to 0), you'll need to delete the *RedBlueCounts* file, as explained in "Managing Memory" on page 207.

REFACTORING

When you *refactor* a program, you make changes to it that increase its quality without affecting how the program behaves. Increasing the quality could mean making the program simpler, easier to understand, or easier to modify. The RedOrBlue-Count program as presented to this point provides a good example of a program that can be improved by simplifying it. Figure 15-25 shows the program's main loop. Notice that within the loop the program displays the values of the variables. This code does exactly the same thing as the code we just added to display the initial values.

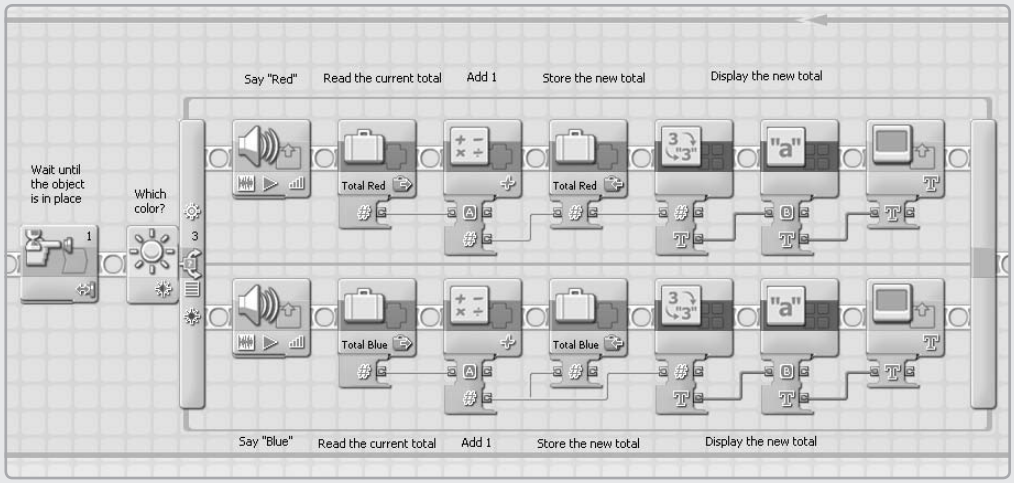


Figure 15-25: The program's main loop

You can eliminate the duplicated code by moving the blocks that display the initial values inside the Loop block. Each time the loop repeats, both values will be displayed on the NXT's screen, eliminating the need to display the value inside the Switch block. Figure 15-26 shows the blocks needed to display the variable values after moving them inside the Loop block, and Figure 15-27 shows the changes to the Switch block.

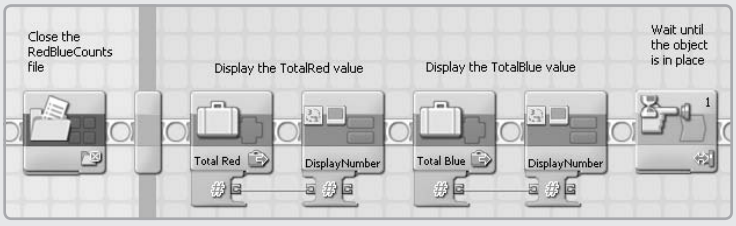


Figure 15-26: Displaying the values inside the loop

When you run the program, it should behave exactly as before. The changes improve the program's quality by making it smaller and removing potential errors that could be caused by displaying the values in two places. This is another example of the DRY (Don't Repeat Yourself) principle.

Refactoring a program as you add new features can help prevent it from growing into an unmanageable mess.

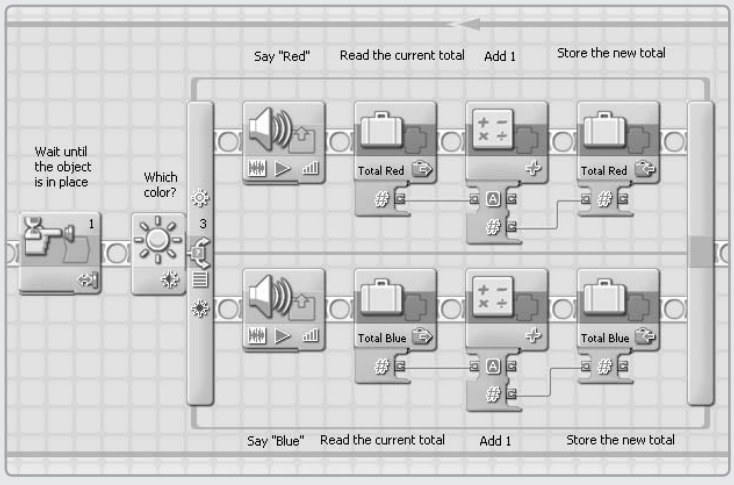


Figure 15-27: Switch block after removing the blocks to display the updated value

managing memory

All the files on your NXT (the programs, sound, images, and data files) take up some of the NXT's limited memory. In this section, I'll show you how to delete files to free up more memory and how to transfer files between the NXT and your computer.

To manage the NXT's memory, click the NXT Window button on the Controller (shown in Figure 15-28), and then select the Memory tab in the NXT window (shown in Figure 15-29).

Figure 15-28: The NXT Window button on the Controller

Figure 15-29: The Memory tab in the NXT window

The left side of the window contains a graphic representation of the NXT's memory, showing the amount used by each type of file and the amount of unused, or free, space. From Figure 15-29 you can see that I've filled about half the space with programs and a smaller amount with sounds and graphics (images). The section labeled *Other* includes the data files created by my programs, which use about a quarter of the NXT's memory. There is still a small portion of unused memory, the exact amount of which is listed on the right side of the window as *Free Storage*. To display individual files, select one of the labels (Program, Sound, Graphic, or Other). In Figure 15-29, the Sound group is selected (the label is slightly bolder than the others), and the center section of the window shows the three sound files used by the programs currently on the NXT.

deleting files

As you write more programs, you'll eventually use all the memory on your NXT. At that point, you'll see the dialog box shown in Figure 15-30 when you try to download a program. To make room for your new program, open the NXT window and delete some files.

Figure 15-30: Out-of-memory error dialog

The Delete All button is the simplest way to free up your NXT's memory. This will delete all the files on the NXT, including any files created by your programs. If you have a data file that you don't want deleted, upload it to your computer before clicking the Delete All button, and then download the file back to the NXT after deleting all the files. Uploading and downloading files is covered in the next section.

Use the Delete button when you want to delete a single file. For example, to delete the *RedBlueCounts* file, follow these steps:

1. Click **Other** to display the data files.
2. Select **RedBlueCounts.txt** from the list of files.
3. Click the **Delete** button.

By default the list of files includes only the ones you've downloaded or created. Check the Show System Files box to include the files used by the demonstration programs and the system sounds (the sounds the NXT makes when you turn it one, download a program, or click a button). Clicking Delete All with this box checked will delete all the system files as well your program and data files, which will free up more room for your programs or for large data files. To keep

the system sounds but delete the demonstration programs, delete individual files. See the "Files and Memory on the NXT" topic in the help file for a list of the system files and a description of each. To restore any system files that you've deleted, you'll need to reload the NXT's firmware, following the instructions in the "Updating the NXTs Firmware" topic in the help file.

transferring files

The Upload button will copy the selected file to your computer. This is useful for saving a backup copy of a data file, perhaps before deleting all the files on the NXT. You may also want to copy a data file to your computer so that you can view or analyze the data using a text editor or spreadsheet program.

Use the Download button to copy a file from your computer to the NXT. The file can be one that you previously uploaded to the NXT or one that you created on your computer. For example, if you write a program that reads commands from a file and performs the specified actions, you could create the list of commands using a text editor on your computer and then download the file to the NXT.

common problems

If you follow the programming instructions for changing the RedOrBlueCount program exactly as given in this chapter, you shouldn't run into any problems. Of course, when writing your own programs, everything won't always work perfectly the first time through, so you'll need to do some debugging. To make finding and fixing bugs go a little faster, here's a list of some of the most common problems that you may encounter when using files:

- **Using the wrong filename.** It's easy to make a mistake when entering a filename. To minimize this problem, copy an existing File Access block instead of adding a new one, or select the filename from the File list in the Configuration Panel.
- **Running out of memory.** File Access blocks that write data will stop working if you use all the NXT's memory. This is often a problem with programs that collect a lot of data. Check for errors when writing data to recognize this situation, so you can then free up some of the NXT's memory.
- **Using too many files at the same time.** Your program can write to or read from up to four files at once. To use a fifth file, you'll first need to close one of the others. For this reason, it's usually best to close a file as soon as you're finished using it.
- **Forgetting to close a file.** After writing data to a file, you must close the file before you can read the data back. You also must close a file you've used before you can delete it.
- **Forgetting to delete a file before writing new data.** The Write Action always adds the new data to the end of the file, so to replace the existing data, you need to first delete the file.
- **Missing data files.** Check for errors when you're reading a program's settings from a data file, and take appropriate action if the file doesn't exist. For example, the RedOrBlueCount program sets the two variables to zero if the *RedBlueCount* file doesn't exist. For some programs, there may not be a reasonable default action to take, and the best you can do is print an error message and stop the program.
- **Mixing up the order of the data.** When you read data from a file, the values will be in the same order used to write them to the file. For example, the RedOrBlueCount program writes the count of the red objects first and then the count of the blue objects. When reading in the initial values, the program uses the same order, setting the Total Red variable first and then the Total Blue variable. If your program uses a data file containing many settings, it's very easy to accidentally switch the order of two items. Be very careful when writing the code to read and write the values to avoid this issue.

conclusion

Files allow you to save data from you program onto the NXT. You can then use that data later in the program, the next time the program runs, or from a different program. The File Access block contains all the features you need to create a file, write and read data, or delete a file.

The Memory tab on the NXT window lets you manage the files on your NXT (either programs or data files). From this window, you can delete files to make room for other programs and transfer files between the NXT and your computer.

16

data logging

The process of acquiring and recording data is called *data logging*. In this chapter I'll show you how to use the NXT-G features you've already learned about to collect sensor data and how to use files to turn the NXT into a *data logger* (a tool for data logging). I'll also introduce you to the special data logging enhancements found in the LEGO MINDSTORMS Education NXT Software 2.0.

data collection and the NXT

The NXT's ability to collect and store data from sensors in a file makes it a useful tool for conducting science experiments. Collecting data is a critical part of performing an experiment, and collecting data by hand can be very tedious and error prone. Most people are just not that good at quickly recording measurements at precise intervals or over long periods of time. Fortunately, these are exactly the kinds of tasks that a computer is good at. The combination of the NXT (a computer) and MINDSTORMS sensors makes the NXT kit a powerful way to create a data logger.

The sensors included with your NXT kit can be used at home or in a science class for a wide variety of interesting experiments. For example, you could use the Light Sensor or Color Sensor to compare the brightness of different brands of compact florescent lights (CFLs) or use the Rotation Sensor to measure area and volume. You can expand the range of possible experiments using the Temperature Sensor from LEGO Education (*http://www.legoeducation.com/*) or one of the many NXT-compatible sensors from HiTech (*http://www.hitechnic.com/*), mindsensors.com (*http://www.mindsensors.com/*), or Vernier (*http://www.vernier.com/*).

Data logging is also a very useful way to learn about the NXT sensors. Sensors play an important part in many robotic programs, so the more you know about how the sensors work, the easier it is to write working programs. When designing a program, it can be useful to first run some experiments to learn how a sensor will react under the conditions you expect your program to experience.

the VerifyLightPointer program

The LightPointer program presented in Chapter 11 uses the Color Sensor or Light Sensor to point the TriBot toward a light source. The TriBot spins in a circle and remembers the position where it detected the brightest light level. After completing a full circle, the TriBot returns to the stored position, which should point it toward the light source.

The design of the LightPointer program depends on two assumptions:

* The sensor can detect the brightest light level while the TriBot is spinning.
* Pointing the TriBot at the brightest detected light level will point it at the light source.

If either assumption is wrong, the program won't work. For example, the program could fail if the robot spins too fast to accurately read the light level or if there is too much ambient light in the room to locate the direction of the light source.

You can verify these two assumptions by conducting an experiment that collects and then analyzes the data from the sensor. The VerifyLightPointer program presented here will collect the data. I'll begin with a very simple program and then expand it.

collecting the brightness data

The LightPointer program continuously reads the Color Sensor or Light Sensor, searching for the highest reading, as the TriBot spins in a circle. The VerifyLightPointer program will do the same thing, except that it will write the sensor readings to a file instead of looking for the highest reading.

Figure 16-1 shows the main part of the VerifyLightPointer program. The Move block starts spinning the TriBot, and then the Loop block continues until the Rotation Sensor for the B motor reads greater than 1100 degrees (which should move the TriBot in a full circle). These two blocks must have the same settings as those in the LightPointer program so that the data is collected using the same conditions as the original program. Figures 16-2 and 16-3 show the Configuration Panels for the Move and Loop blocks.

NOTE **For the balloon tires, you can use 800 degrees for a full circle.**

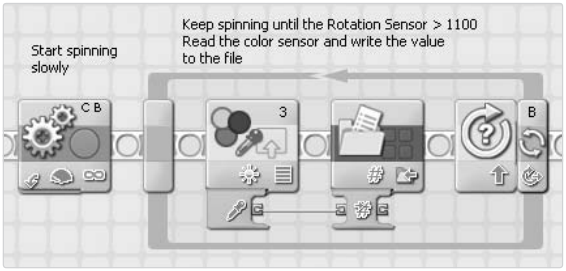


Figure 16-1: Saving the Color Sensor data

Figure 16-2: Spinning the TriBot slowly

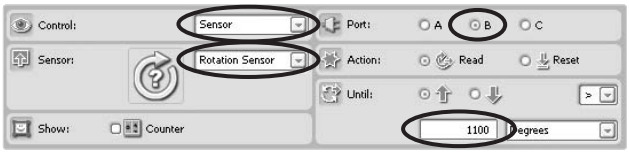


Figure 16-3: Looping until motor B reaches 1100 degrees

Inside the Loop block, the Color Sensor block uses Light Sensor mode, with the light turned off to read the amount of ambient light, as shown in Figure 16-4. When using the Light Sensor, the Generate light option should be turned off (see Figure 16-5).

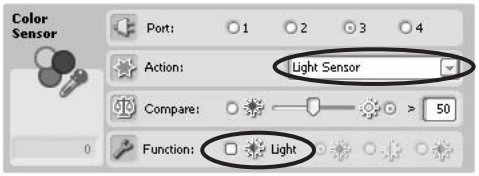


Figure 16-4: Reading the amount of ambient light using the Color Sensor

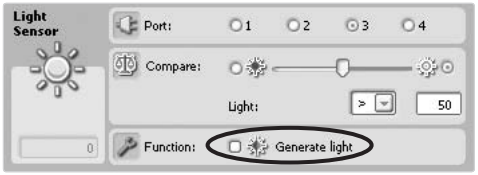


Figure 16-5: Reading the amount of ambient light using the Light Sensor

The sensor reading is passed to the File Access block, which writes the number to the file *VLPData*. Figure 16-6 shows the Configuration Panel for this block.

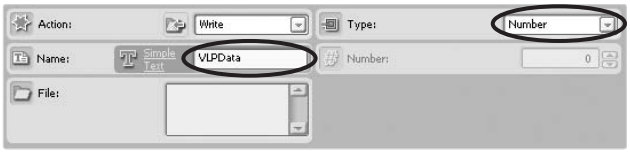


Figure 16-6: Writing the sensor reading to the VLPData *file*

To finish the program, you need to add some housekeeping blocks to get everything ready at the beginning of the program and to clean things up at the end, as shown in Figure 16-7.

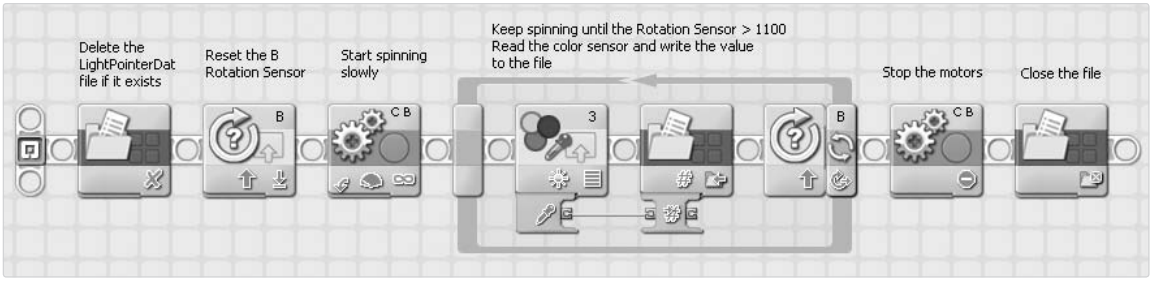


Figure 16-7: The completed program to collect the light level data

The program begins with a File Access block (see Figure 16-8), which deletes the *VLPData* file if it exists. If you omit this block, the program will work as expected the first time you run it, but after that, it will keep adding the new data to the end of the file (after the data from the previous run), instead of replacing it.

Figure 16-8: Deleting the VLPData *file*

The Rotation Sensor block (see Figure 16-9) resets the B motor sensor to zero to ensure that the TriBot spins for the entire 1,100 degrees.

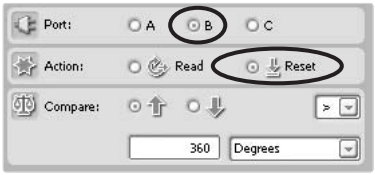


Figure 16-9: Resetting the Rotation Sensor for the B motor

The two blocks at the end of the program stop the motors and close the file. These blocks aren't strictly necessary because when the program ends, the motors will stop, and the file will be closed. I added them here because it's good programming practice to always clean up; it makes the code more reusable and helps you avoid bugs if you decide to add more code to the program in the future. Figures 16-10 and 16-11 show the Configuration Panels for the Move and File Access blocks.

Figure 16-10: Stopping the motors

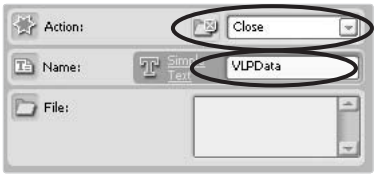


Figure 16-11: Closing the VLPData *file*

running the program

When you run this program, it will create a fairly large data file. To be sure that you don't run out of memory while collecting the data, delete all the files on the NXT (see "Deleting Files" on page 207) before downloading and running the program. Then, position the TriBot and a light as shown in Figure 16-12, and run the program. The robot should spin slowly in a circle and then stop.

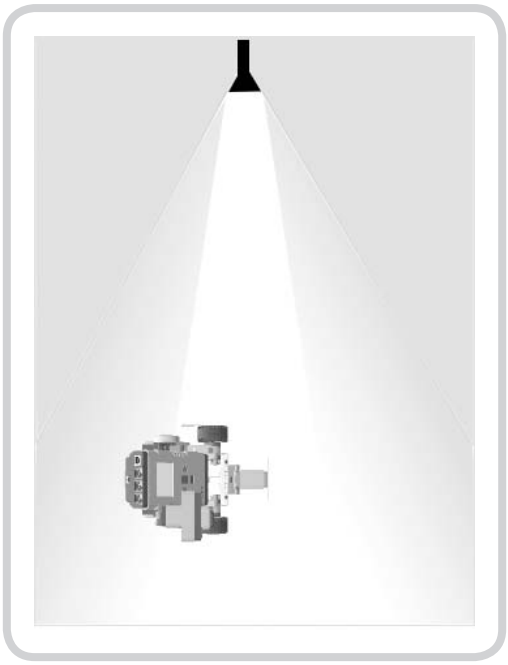
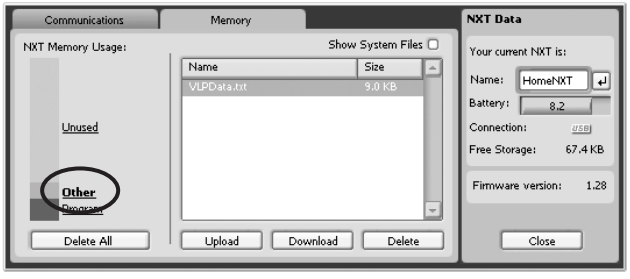


Figure 16-12: The starting position

analyzing the data

After running the program, you should find a data file named *VLPData.txt* on the NXT. The next step is to move the file to your computer and examine the data. Follow these steps to upload the file:

1. Click the NXT Window button on the Controller.
2. Select the Memory tab in the NXT window, and then select **Other** from the NXT Memory Usage section. *VLPData.txt* should be the only file listed.



3. Click the Upload button, and you'll be prompted to select the folder on your computer where the file should be placed.

You can open the file in a text editor, word processor, or spreadsheet program. I prefer to use a spreadsheet (such as OpenOffice.org Calc or Microsoft Excel) to analyze data so I can both look at the raw numbers and easily create graphs. Figure 16-13 shows a graph of the measurements taken during my test run. The light level increases significantly as the robot turns toward the flashlight and forms a nice peak. Based on this data, it seems that the LightPointer program should be able to correctly identify the direction of the light source.

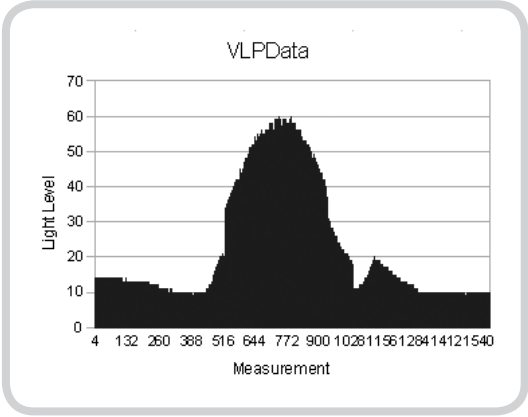


Figure 16-13: Graph of the light level data

adding rotation sensor data and a timestamp

Although this data looks promising, you may want to collect two additional pieces of information: the Rotation Sensor reading and a timestamp. Since the LightPointer program uses the Rotation Sensor to remember the robot's position, it's a good idea to record its reading in addition to the light level. A *timestamp* is a value that indicates when a measurement was taken. One of the NXT's timers can be used to create a timestamp. You can use timestamps to tell when each measurement was taken and how often the measurements are recorded.

Instead of using a File Access block for each item (the timestamp, the light level, and the Rotation Sensor reading), you'll create one text value that contains the three numbers separated by commas. The data file created by the program will contain one line for each measurement. Each line will contain all three pieces of information separated by commas, a file format known as *comma-separated values*. Spreadsheet programs know how to work with data files in this format, so arranging the data this way will make it easier to analyze.

The concept behind formatting the data is straightforward. You'll use a Timer block, a Rotation Sensor block, and a Color Sensor block to acquire the values; Number to Text blocks to convert the three numbers to text values; and Text blocks to join the values together separated by commas.

Although the concept is simple, this process requires several blocks connected by data wires. Listing 16-1 describes this process using pseudocode, and Figure 16-14 shows the new blocks added to the program.

```
read the Timer
convert the value to text
read the Rotation Sensor
convert the value to text
read the light level from the Color Sensor
convert the value to text
use a Text block to combine the Timer value, a
   comma, and the Rotation Sensor value
use a Text block to add another comma and the value
   from the Color Sensor
```

Listing 16-1: Combining the timestamp, motor rotation, and light level values

The Timer block (Figure 16-15) uses all the default settings to read Timer 1. The Rotation Sensor block reads the position of motor B (Figure 16-16). The settings for the Color Sensor (or Light Sensor) block don't need to change.

Figure 16-15: Reading the Timer 1 value

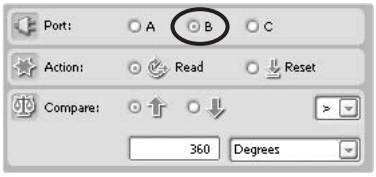


Figure 16-16: Reading the position of motor B

Both Text blocks have the same configuration (shown in Figure 16-17). Data wires supply the A and C values, and the B value is set to a comma in the Configuration Panel. The first Text block will join the timer value with the Rotation Sensor value, and the second Text block will add the light level value.

For example, assume the Timer block value is 5, the Rotation Sensor block value is 10, and the Color Sensor value is 15. First, the three values are converted to text by the Number to Text blocks. Then the input to the first Text block will be 5 and 10, and the output value will be `5,10`. This value is then passed to the second Text block, so its input values will be 5,10 and 15, and its output value will be `5,10,15`.

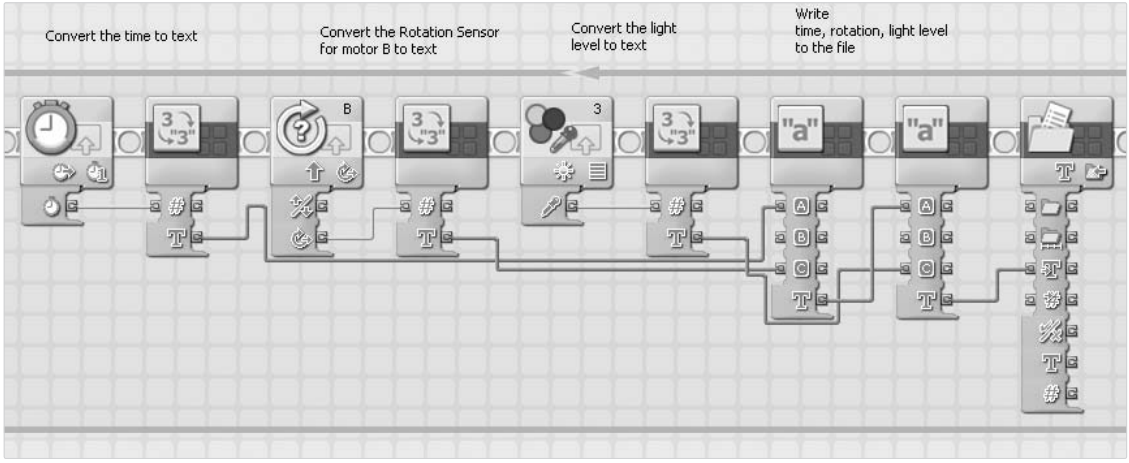


Figure 16-14: Recording the timestamp, rotation, and light level

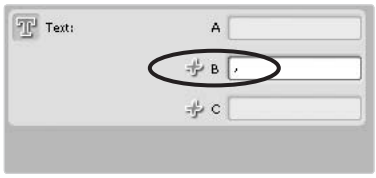


Figure 16-17: Joining the two input values with a comma

You also need to change the Type setting on the File Access block because you are now writing out text instead of a number (shown in Figure 16-18).

Figure 16-18: Writing a text value to the file

There are two more blocks to add to the beginning of the program, as shown in Figure 16-19. The File Access block adds `Time, Rotation, Light` as a header line to the file (see Figure 16-20) to make it easier to tell what each value means when analyzing the file. The Timer block simply resets Timer 1, as shown in Figure 16-21.

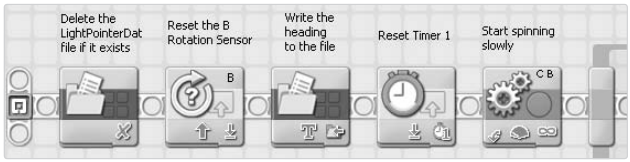


Figure 16-19: Text and Timer blocks added before the Move block

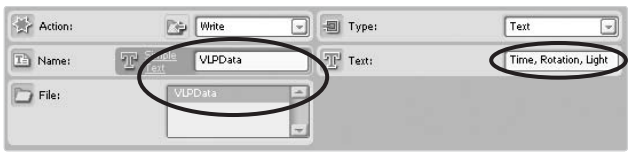


Figure 16-20: Writing a header line to the file

Figure 16-21: Resetting Timer 1

Run the program again, and the *VLPData* file will be replaced, only this time it will contain a timestamp and the position of motor B in addition to the light level. Upload the file to your computer, open it in a spreadsheet program, and you should see three columns of data, as shown in Figure 16-22.

	A	B	C
1	Time	Rotation	Light
2	3	0	17
3	175	5	17
4	179	6	17
5	184	6	18
6	188	7	18
7	192	7	17
8	196	8	17
9	201	8	18
10	205	9	17

Figure 16-22: Three columns of data

gaps in the data

Using the new data, you can graph the relationship between the position of motor B and the light level. Figure 16-23 shows an X/Y (or scatter) plot using the Rotation and Light columns.

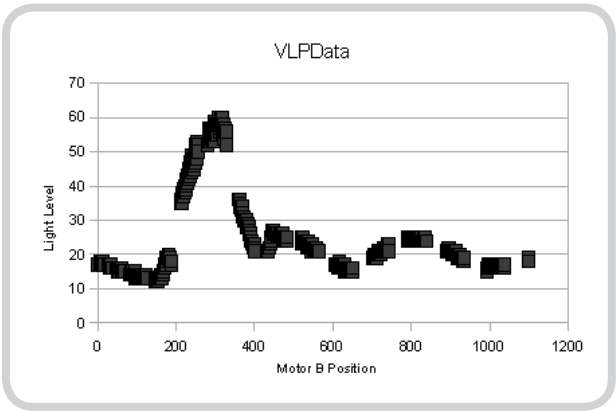


Figure 16-23: Graph of the light level and motor B position

Although this graph has the same general shape as the one shown in Figure 16-13, there are gaps in the data. Figure 16-24 shows the data at one of these gaps. The time values are in milliseconds (thousandths of a second), and you can see that there is usually a four- or five-millisecond interval between each reading (each timestamp increases by four or five). However, the timestamps for lines 256 and 257 show a 180-millisecond gap, which shows up in the graph as a break in the data.

	A	B	C
1	Time	Rotation	Light
253	1738	188	17
254	1742	188	17
255	1746	189	17
256	1751	189	18
257	1930	213	35
258	1934	213	36
259	1939	214	35
260	1943	215	36

Figure 16-24: Time, Rotation, and Light values at a gap in the graph

The gaps are caused by the way the NXT allocates, or sets aside, memory for a file. When a program first creates a file, the NXT allocates a small amount of memory for the file. As the program writes to the file, it will eventually fill all the memory allocated for that file, at which point the NXT will allocate a larger section of memory and copy all the data in the file to the new location. The time taken to copy the file to a larger section of memory causes the gaps in the data collection.

For large files, like the one we're working with, the NXT may go through this reallocation process several times. You can avoid this problem by making the file large enough when it's first created. The File Access block has a Initial File Size data plug that allows you to specify how big to make the file. Follow these steps to get a starting value for the file size:

1. Use the NXT window to delete all the files on the NXT.
2. Download (but don't run) the VerifyLightPointer program using the Controller's Download button, shown here:

3. Open the NXT window to see how much free space is available.

From Figure 16-25 you can see that there are 82.3KB of free space (a kilobyte is a little more than 1,000 bytes). You may see a slightly different number if you're using a different version for the software. Setting the file size to 70KB will use most of the free space and leave a little extra space in case you decide to expand the program.

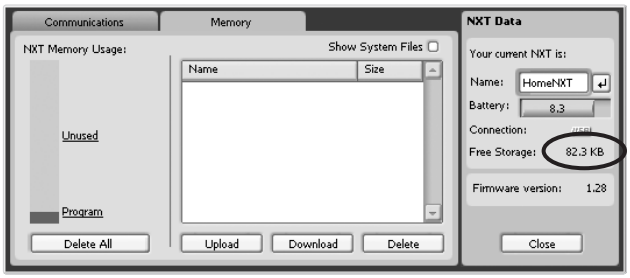


Figure 16-25: Free memory after downloading the VerifyLightPointer program

setting the initial file size

To set the size of the file, you need to use a data wire to pass a value to the File Access block's Initial File Size data plug. This is one of the few settings that doesn't appear on the Configuration Panel, and you can set it only using a data wire. The value passed to the Initial File Size data plug is used only when the File Access block creates the file, which happens if the Action option is set to Write and the file doesn't already exist. In the VerifyLightPointer program, the block that writes the heading to the file also creates the file, so this is the block you need to use to set the file size. The Initial File Size data plug expects the value to be in bytes, so you'll set the value to 70000 using a Math block as follows:

1. Add a Math block before the File Access block that writes the heading to the file.
2. In the Math block's Configuration Panel, set the A value to **70000**.
3. Connect the Math block's Result data plug to the File Access block's Initial File Size data plug.

Figure 16-26 shows the changes to the program, and Figure 16-27 shows the Configuration Panel for the Math block.

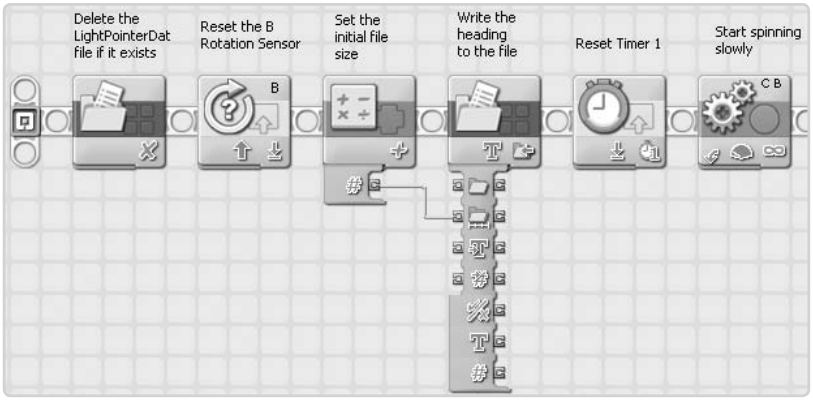


Figure 16-26: Setting the initial file size

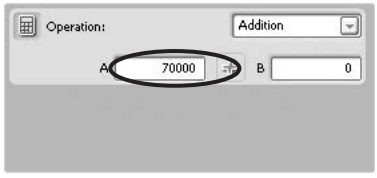


Figure 16-27: The Math block's Configuration Panel

Now run the program again, and transfer the file to your computer. Figure 16-28 shows the graph of the data for my test. There are no big gaps in the data, and the light level forms one well-defined peak. Based on this data, I have a high degree of confidence that the LightPointer program will be able to find the direction of the light source.

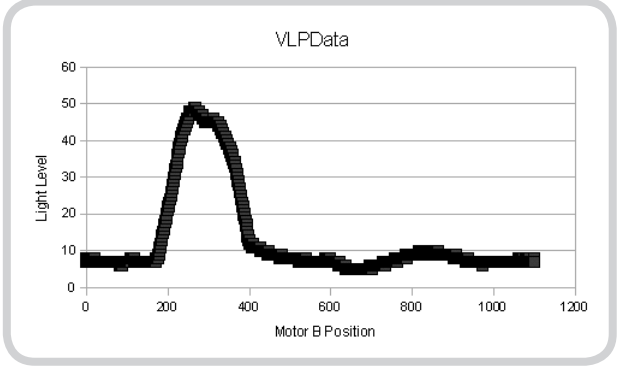


Figure 16-28: The data without gaps

controlling the amount of data

The VerifyLightPointer program is a little unusual in that it collects and records data as fast as possible, creating a large data file in a short amount of time. The program is written this way to match the design of the LightPointer program, but for most data-logging programs, you'll want more control over how often the program records data. Because the amount of memory you have to work with is limited, you'll usually need to record the data less often in order to avoid running out of memory.

Most data-logging programs will be structured like the VerifyLightPointer program. After some blocks perform the initial setup, a Loop block will contain the code to collect the data and write it to a file. You can control how often the data is recorded by adding a Wait block at the end of the body of

the Loop block. How long should the Wait block pause? That depends on how long you expect the experiment to take and how often the data you're collecting changes. You need to record the data often enough that you don't miss any important changes but not so often that you run out of memory. Finding the right balance often involves some trial and error, so don't be surprised if you need to change the settings a few times to get them just right.

For example, say you decide to change the VerifyLightPoint program so that it takes 20 measurements per second. To do this, you'll add a pause at the end of the Loop block to wait for one twentieth of a second, or 0.05 seconds. Figure 16-29 shows the Wait block added to the program, and Figure 16-30 shows the block's Configuration Panel.

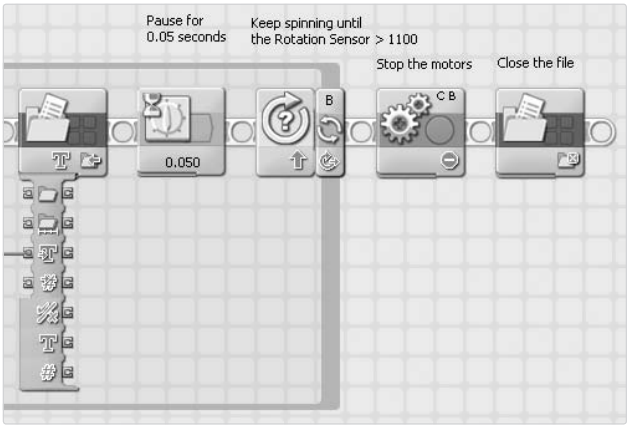


Figure 16-29: A Wait block added at the end of the Loop body

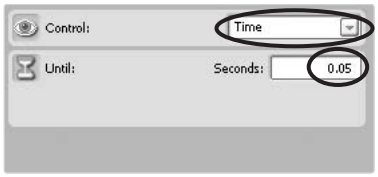


Figure 16-30: Waiting for 0.05 seconds

Now when you run the program, it will pause for one-twentieth of a second (0.05 seconds) each time through the loop, which means that the data will be recorded at a rate of about 20 readings per second. (There won't be exactly 20 readings per second because it takes a bit of time to control the motors, collect the readings from the sensors, and write the data. The time between readings will be slightly more than the 0.05-second pause from the Wait block, but it will be close enough for our purposes.)

data logging using the LEGO MINDSTORMS education NXT software 2.0

The MINDSTORMS Education NXT Software 2.0 contains some features to support data logging that are not available in the other software releases. In this section I'll briefly cover the data-logging blocks and the NXT data-logging application. (In-depth coverage of these topics is beyond the scope of this book; to learn more, see Damien Kee's "Datalogging Activities for the Busy Teacher" at *http://www.domabotics.com/books/*.)

the data-logging blocks

The Start Data Logging and Stop Data Logging blocks, in the Advanced group on the Complete Palette (shown in Figure 16-31), contain all the functionality you need for most data-logging activities. As you'll see, these two blocks will allow you to rewrite the VerifyLightPointer program using only three blocks.

Figure 16-31: The Start Data Logging and Stop Data Logging blocks

The Configuration Panel for the Start Data Logging block (shown in Figure 16-32) contains all the controls you need to collect data from up to four sensors:

- The Name setting controls the name of the data file to create.
- The Duration item controls how long to collect data in either seconds or minutes. The Duration can also be set to Single Measurement to take one reading or Unlimited to continue recording data until a Stop Data Logging block is used.
- The Rate setting controls how often to save a measurement by setting either the number of samples to take per second or the number of seconds between each sample.
- The Wait for Completion option lets you decide whether the program should continue while collecting data.

The controls on the right side of the Configuration Panel let you select the type of sensor to use and the port the sensor is attached to. Once you select a sensor type, additional controls appear to let you select the measurement unit to use or control the light on the front of the Light Sensor (see Figure 16-33). A timestamp is automatically added to the data collected, so you don't need to use a timer as you did in the VerifyLightPointer program.

The Configuration Panel for the Stop Data Logging block has only one setting: the name of the data file to stop using (as shown in Figure 16-34). This should be the same name you used in the Start Data Logging block. You need to use a Stop Data Logging block only if you choose Unlimited for the Duration setting of the Start Data Logging block.

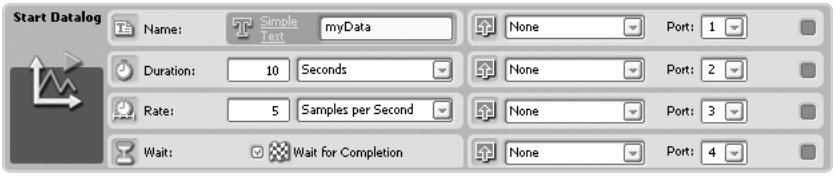


Figure 16-32: The Configuration Panel for the Start Data Logging block

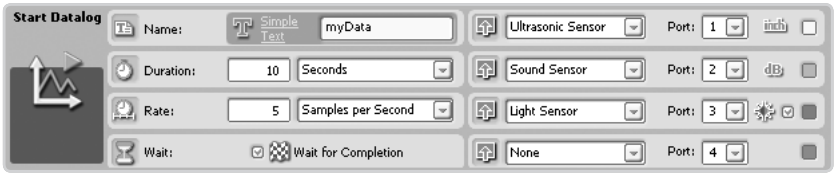


Figure 16-33: Controls for setting sensor options

Figure 16-34: The Configuration Panel for the Stop Data Logging block

the VerifyLightPointer2 program

Most of the blocks in the VerifyLightPointer program are used to collect, format, and record the data from the sensors. These blocks can be replaced by a single Start Data Logging block. The VerifyLightPointer2 program, shown in Figure 16-35, uses the data-logging blocks to collect the Light and Rotation Sensor data using just three blocks. (This program uses the Light Sensor because the Color Sensor was not available when the Education Software 2.0 was released.)

Figure 16-36 shows the Configuration Panel for the Start Data Logging block, which does most of the work in this program. (Because most of the settings have been changed from the defaults, the changes aren't

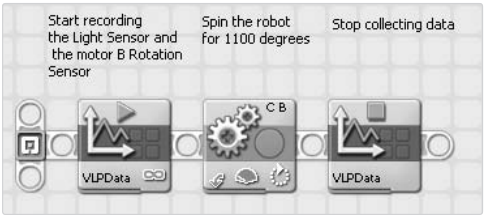


Figure 16-35: Using the data-logging blocks

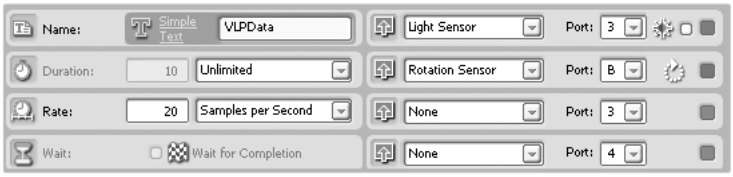


Figure 16-36: The Start Data Logging block's Configuration Panel

highlighted.) When this block runs, it first creates a file named *VLPData.log*. It will then start collecting data from the Light Sensor and Rotation Sensor at a rate of 20 samples per second. The Duration is set to Unlimited, so the data collection will continue until a Stop Data Logging block is used or the program ends.

The Move block spins the robot around in a complete circle using the settings shown in Figure 16-37. The Start Data Logging block will continue collecting data while the Move block runs, so you don't need to use a Loop block like in the original program.

Figure 16-37: Spinning the robot for 1100 degrees

Finally, the Stop Data Logging block (shown in Figure 16-38) ends the data collection once the Move block has finished.

Figure 16-38: Stopping the data collection

When you run this program, it will spin the TriBot in a circle and record the readings from the Light Sensor and Rotation Sensor, automatically adding a timestamp to each reading. The data is stored in a file named *VLPData.log*, which you can transfer to your computer and open in a spreadsheet program. You can also view the data using the NXT Data Logging application discussed in the next section.

NOTE **The Start Data Logging block uses a tab character instead of a comma to separate the data, so if your spreadsheet program doesn't automatically convert the file, you may need to select Tab as the delimiter.**

the NXT data logging application

The NXT Data Logging application is an easy-to-use tool for analyzing the data collected by the Start Data Logging block. You can view and graph data from logfiles stored on your computer or loaded from the NXT. For example, Figure 16-39 shows a graph of the Light Sensor readings collected by the VLPData2 program.

You can also create a very simple program using the NXT Data Logging application. The Experiment Configuration window (shown in Figure 16-40) essentially creates a program consisting of a single Start Data Logging block. You can download, start, and stop the program and upload the data using the NXT Data Logging application, making it easy to conduct many science experiments without having to write a new program or switch between the regular MINDSTORMS application and the Data Logging application.

Perhaps the best feature of the Data Logging application is that it can collect and graph the data while the program is running, which allows you to see how the data looks while an experiment is in progress. When coupled with a projector and a class full of inquisitive students, this can be a great tool for creating an interactive learning experience.

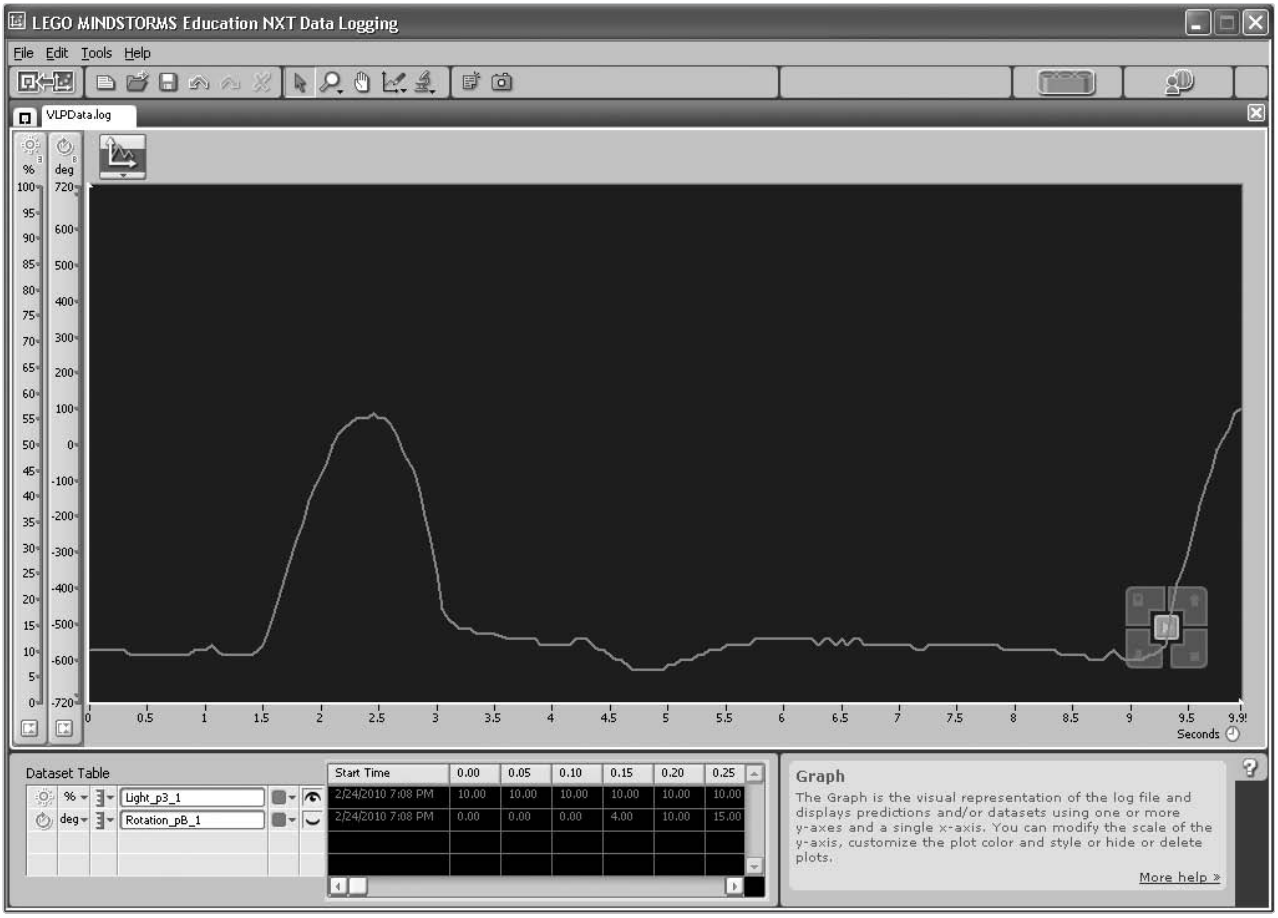


Figure 16-39: The NXT Data Logging application

Figure 16-40: The Experiment Configuration window

conclusion

Turning the NXT into a data logger is easy because of its ability to collect, format, and record data from a variety of sensors. Data logging can be useful as part of a classroom science experiment that may have little or nothing to do with robotics or as a way to learn more about how the NXT sensors work. The VerifyLightPointer program contains all the steps you'll need in a typical data-logging program. These include creating the data file, collecting the sensor data, writing the data to the file with a timestamp, and controlling the rate of the data collection.

The MINDSTORMS Education NXT Software 2.0 contains two features that make data logging with the NXT even easier. The Start Data Logging and Stop Data Logging blocks can perform the same tasks that require several Sensor, File Access, and Text blocks. The NXT Data Logging application lets you analyze the data collected and even view the data while the experiment is in progress.

17

using multiple sequence beams

In this chapter you'll learn how to use more than one Sequence Beam in your programs, allowing your robot to do more than one thing at a time. I'll start by showing you how to add a simple odometer to the AroundTheBlock program and then move on to a more complicated project, adding flashing lights to the DoorChime program. I'll also discuss a few complications to the rules concerning program flow that arise when you use multiple Sequence Beams. Then I'll show you how to use these rules to synchronize the actions between Sequence Beams.

multitasking

Multitasking means doing more than one thing at a time. You use multitasking in a program to make your robot perform two (or more) independent tasks simultaneously. For example, your program could have one section to control the robot's navigation and a separate section to collect sensor data.

In NXT-G, multitasking is accomplished using multiple Sequence Beams. Figure 17-1 shows an example based on the AroundTheBlock program introduced in Chapter 4. The blocks on the Sequence Beam at the top of the program move the TriBot around a square, and the blocks on the Sequence Beam at the bottom continually display the motor position.

When you run this program, the NXT will start both Loop blocks and then rapidly switch between running the code on each Sequence Beam. The computer inside the NXT can't really do more than one thing at a time, but it can switch between the two tasks quickly enough to perform both tasks successfully. The result is that the TriBot moves around the square while the display shows how far the robot has moved.

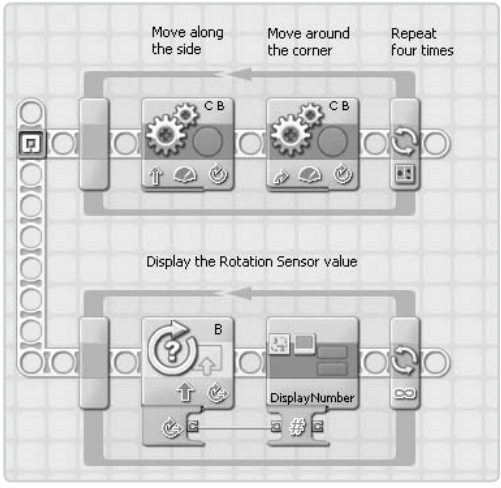


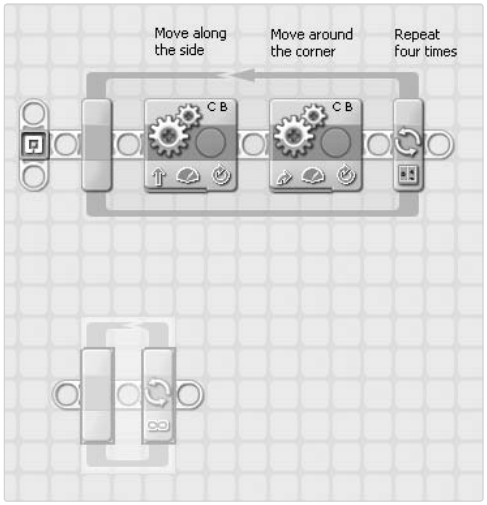
Figure 17-1: Displaying the motor position while moving around a square

adding a second sequence beam

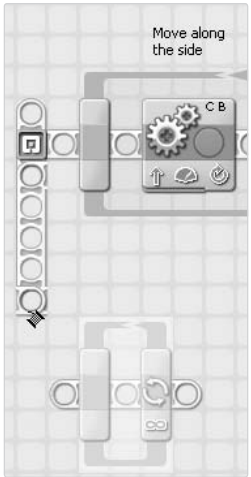
In this section I'll walk you through the steps to add the second Sequence Beam to the AroundTheBlock program. As you're adding blocks and connecting the Sequence Beam, be sure to go slow and give the IDE time to keep up with you. Moving too fast is the most common source of problems people have when editing complex programs. As always, select the Edit ▸ Undo

menu item if you move too fast or make a mistake. Follow these steps to add a Loop block on a new Sequence Beam:

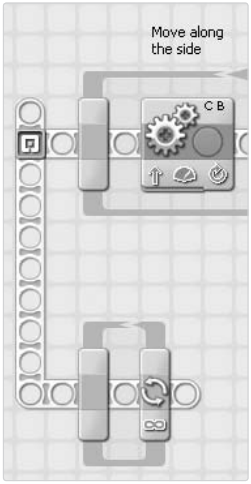
1. Open the AroundTheBlock program.
2. Place a Loop block in the Work Area below the existing program. The block will appear faded because it's not connected to a Sequence Beam.



3. Move your mouse over the downward-pointing beam at the starting point of the program. The mouse cursor should change to the wire spool (the same image that's used when drawing data wires).
4. Click the mouse button, and then drag the beam down toward the Loop block. The Sequence Beam should grow as you move the mouse.

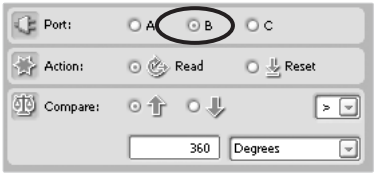


5. Drag the Sequence Beam down and to the right to connect it with the Loop block, and click the mouse button. Once the Loop block is connected to the Sequence Beam, it should no longer appear faded.

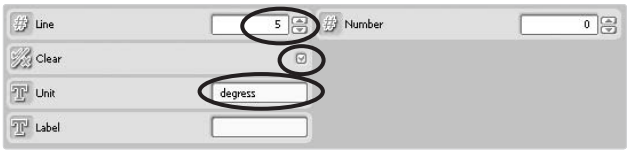


With the Loop block in place, you can add the Rotation Sensor and DisplayNumber blocks. (The DisplayNumber block is a My Block that I presented in Chapter 13.)

6. Add a Rotation Sensor block to the Loop block. Configure the block to read the position of motor B.



7. Add a DisplayNumber block, and set it to clear the screen and display the value on line 5. Enter **degrees** for the Unit setting.



8. Draw a data wire between the Rotation Sensor block's Degrees data plug and the DisplayNumber block's Number data plug. Close the data hubs on these two blocks. The program should now look like Figure 17-1.

Now run the program, and observe how it behaves. You should see the position displayed as the TriBot moves around the square. Once the TriBot has completed moving around all four sides of the square, the program continues to display the position of motor B. Pick up the robot and turn the motor, and you'll see the position updated on the screen.

The original AroundTheBlock program ends after the robot moves around the square because the Loop block finishes after repeating four times and there are no more blocks on the Sequence Beam. When more than one Sequence Beam is used, the program continues to run until it reaches the end of all the Sequence Beams. Since the Loop block that displays the motor position is set to run forever, the program will continue running until you stop it.

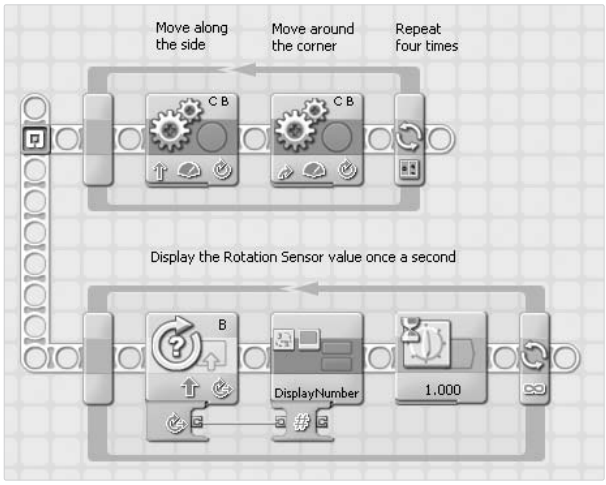


Figure 17-2: Slowing down the busy loop

avoiding a busy loop

If you watch closely, you may notice that after you add the code to display the motor position, the AroundTheBlock program doesn't get the TriBot back to the starting point as accurately as the original program did. The problem is that the Move blocks on the top Sequence Beam are a little less accurate in the new program because the NXT is spending about half its time displaying the motor position.

The code to display the motor position is an example of a *busy loop*, one that repeats as fast as possible and consequently uses a large portion of the NXT's processing power. You can improve the accuracy of the Move blocks by slowing down the display loop, which will allow the NXT to spend more of its time making sure the movements are accurate. Simply adding a one-second pause to the Loop block, as shown in Figure 17-2, will improve the program's accuracy noticeably. Updating the display once a second (instead of as fast as possible) doesn't detract at all from the usefulness of the displayed value.

adding a sequence beam to a loop block

Adding a Sequence Beam to a Loop or Switch block requires some additional steps because you'll need to make room for the new blocks. In this section, you'll modify the DoorChime program from Chapter 13 (shown in Figure 17-3) to flash the light on the Color Sensor or Light Sensor while playing the chime.

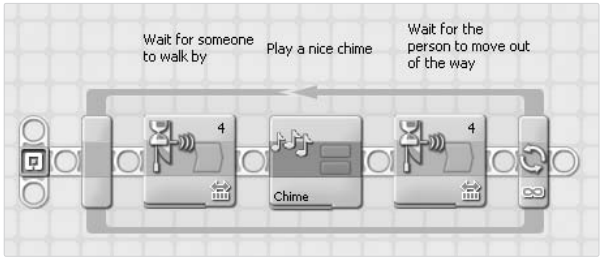


Figure 17-3: The DoorChime program

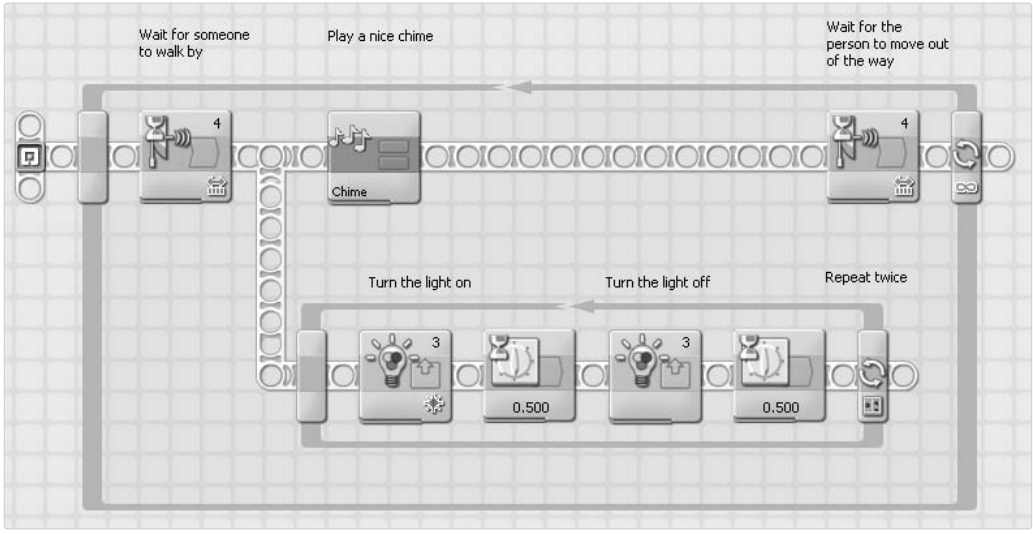


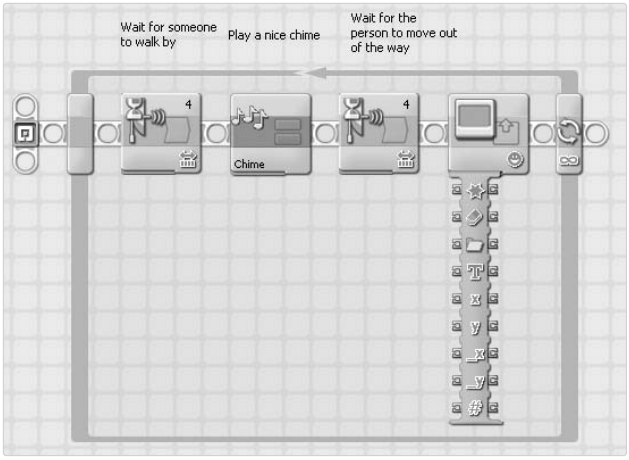
Figure 17-4: Flashing the light while playing the chime

Figure 17-4 shows how the program will look when the changes are completed. The blocks on the second Sequence Beam flash the light on the Color Sensor by turning the light on and off with a half-second pause between each change. The Chime block takes two seconds to play all the notes, so to make the light flash for the same amount of time, the Loop block is set to repeat twice.

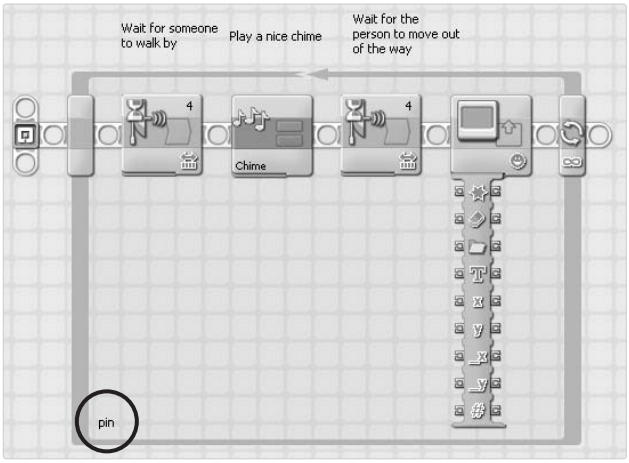
the crowbar and pin technique

You'll begin by making changes similar to those you made for the AroundTheBlock program; you'll add a Loop block and then connect the Sequence Beam to the new block. However, there isn't room within the Loop block to add a new block below the Sequence Beam, so you first need to expand the Loop block so that there's room for the new code. To do so, follow these steps, known as the *crowbar and pin* technique.

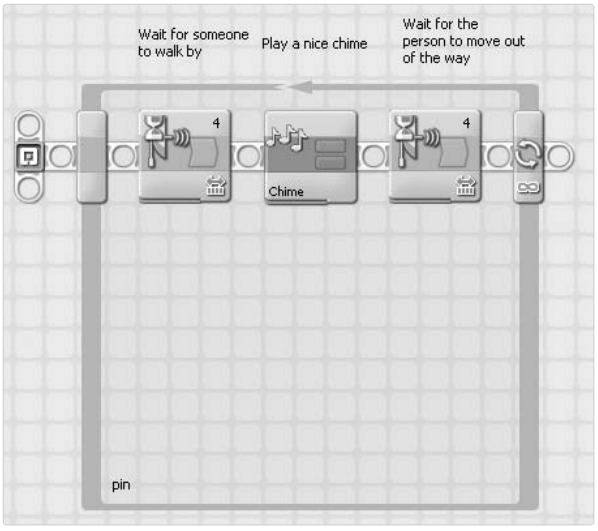
1. Add a Display block to the end of the Loop block, and open its data hub. This is called the *crowbar*, because it pries open the Loop block.



2. Add a comment to the bottom-left side of the Loop block. This is called the *pin*, because it will keep the Loop block from closing up when the Display block is removed. The text of the comment isn't important; I used *pin* here to make the purpose clear.



3. Delete the Display block.

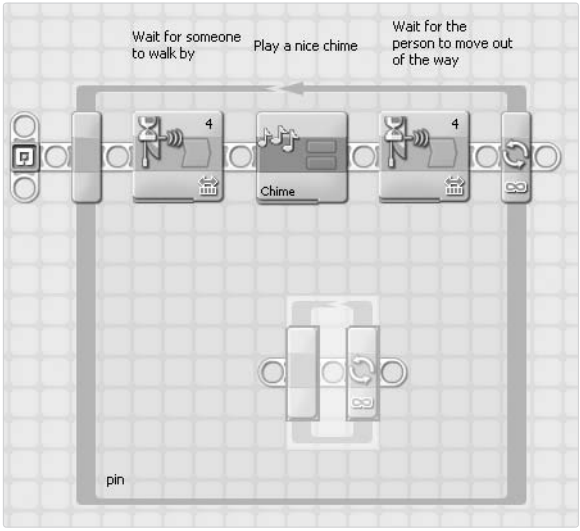


The comment should keep the Loop block from closing, leaving you with room to add the new code. I used a Display block as the crowbar because it has a long data hub and therefore makes the Loop block grow enough to add another Sequence Beam. The crowbar and pin technique is also useful if you want extra room for drawing data wires between the blocks within a Loop or Switch block.

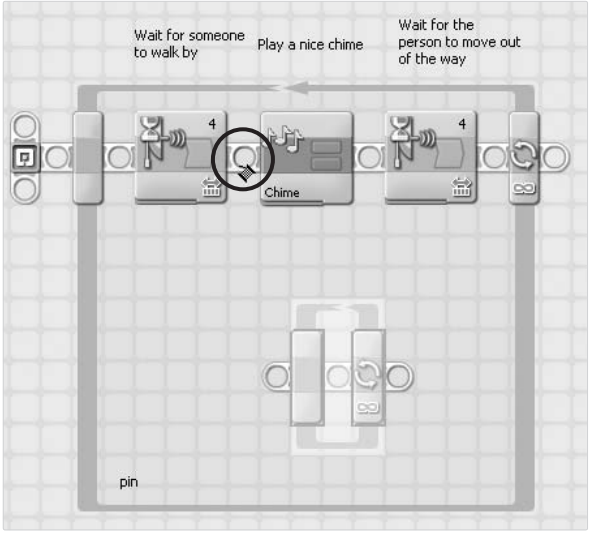
adding the sequence beam

Now that there's room in the Loop block, you can add the new code. Follow these steps to add a new Loop block and connect it to a new Sequence Beam:

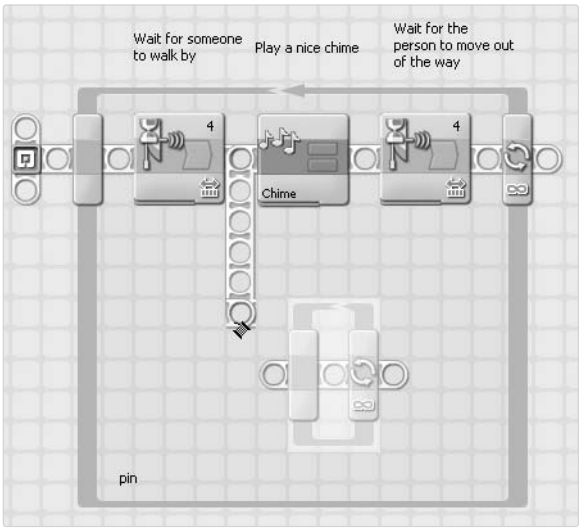
1. Add a new Loop block in the empty space below and a little to the right of the Chime block.



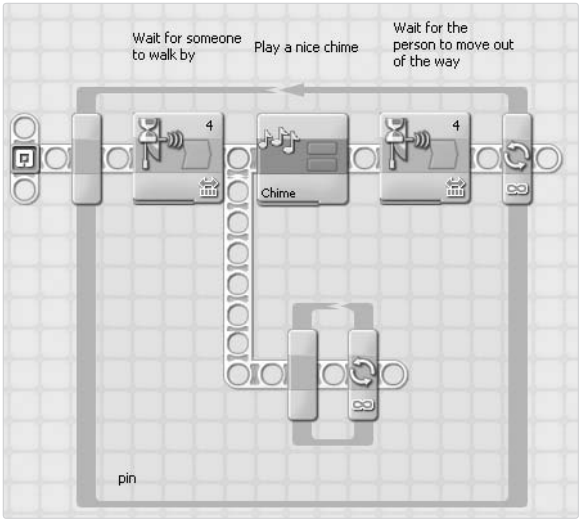
2. Move your mouse over the Sequence Beam to the left of the Chime block. Hold down the SHIFT key, and click the mouse button. The mouse cursor should change to the wire spool (the image used when drawing data wires).



3. Move the mouse down, and a new Sequence Beam should follow the mouse.



4. Connect the Sequence Beam to the Loop block.



expanding the loop block

As you add blocks to the new Loop block, it will grow wider. Eventually it will reach the edge of the outer Loop block, and unfortunately the outer Loop block won't automatically grow wider. If you try to add more than one block to the inner Loop block, the Sequence Beam behaves oddly, and the right side of the Loop block gets cut off, as shown in Figure 17-5.

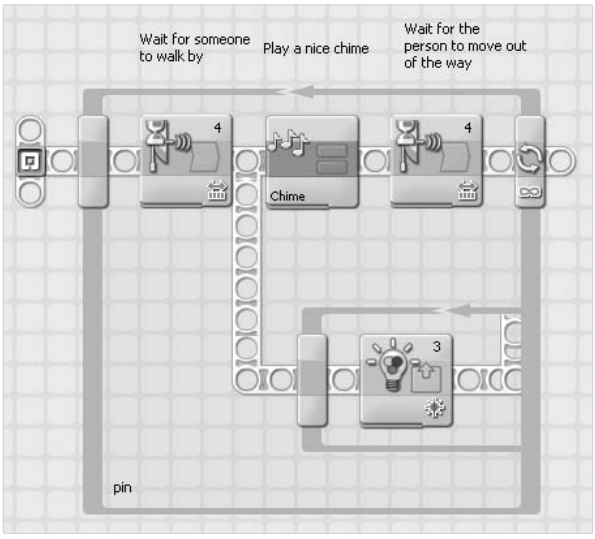
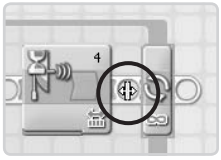


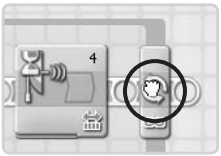
Figure 17-5: After adding two blocks to the inner Loop block

To make room for more than one block, you need to make the outer Loop block wider, following these steps:

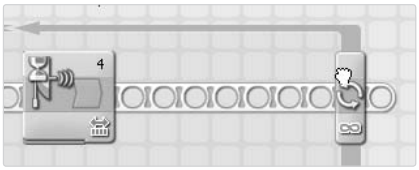
1. Select the outer Loop block.
2. Move the mouse over the Sequence Beam between the Ultrasonic Sensor block and the right edge of the outer Loop block.



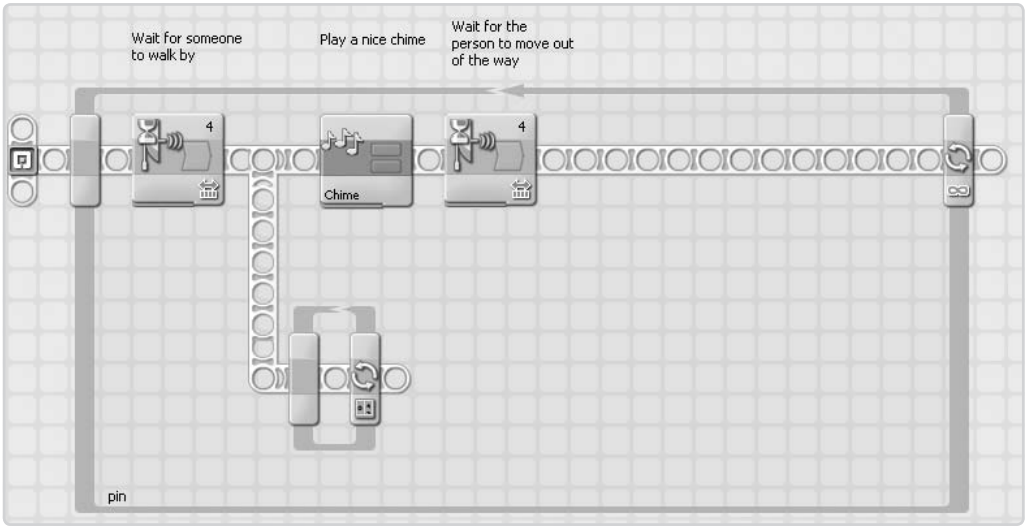
3. Hold the mouse button down, and slowly move the mouse to the right. The mouse cursor should change to a hand as it moves over the edge of the Loop block. If the mouse cursor doesn't change to a hand as you move over the Loop block, press the ESCAPE key, select the Loop block again, and start over.



4. As you drag the mouse to the right, the Loop block should get wider.



5. Drag the edge of the Loop block to the right so that the program looks like this:



As you add blocks to the inner Loop block, you might find that the outer Loop is either too wide or too narrow. You can use the technique shown in this section to adjust the width of the Loop block later as needed.

making the light flash

To control the light on the Color Sensor, use the Color Lamp block in the Action group on the Complete Palette (shown in Figure 17-6). The Configuration Panel, shown in Figure 17-7, allows you to turn the lamp on and off and select the color (red, green, or blue).

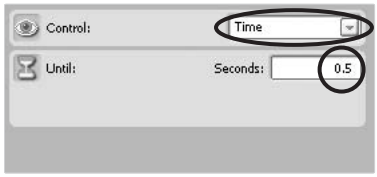
Figure 17-6: The Color Lamp block

Figure 17-7: The Color Lamp block's Configuration Panel

To control the light on the Light Sensor, use the Light Sensor block.

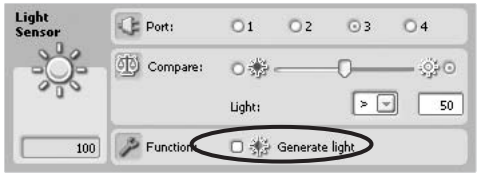
Follow these steps to make the light turn on and off:

1. Add a Lamp block or a Light Sensor block to the inner Loop block, depending on which sensor you're using. Keep all the default settings, because these will turn the red light on.
2. Add a Wait block. Set the Control item to **Time**, and set the delay to **0.5** seconds.

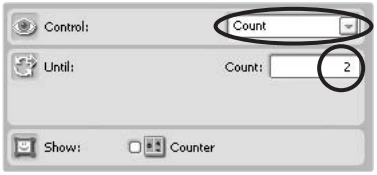


3. If you're using the Color Sensor, add another Lamp block to turn the light off.

4. If you're using the Light Sensor, add another Light Sensor block to turn the light off.



5. Add another Wait block. Set the Control item to **Time**, and set the delay to **0.5** seconds. The Configuration Panel should look exactly like the one for the first Wait block.
6. Select the inner Loop block, and set it to repeat twice.



At this point, the program should look like this:

The Chime block and the new code to flash the light will each take two seconds, so they should finish at at about the same time; once they finish, the second Ultrasonic Sensor block will run. You can make this clearer by dragging the Ultrasonic Sensor block to the right so that it's at the end of the loop, as shown in Figure 17-8. It's important to understand that moving the block along the Sequence Beam like this doesn't change how the program behaves; it simply makes the program visually match the way you expect it to behave. I'll show you how to make sure the program works correctly in "Synchronizing Two Sequence Beams" on page 230.

Now run the program, and when someone walks by the TriBot, you should see the light flash on and off twice while the chime is played.

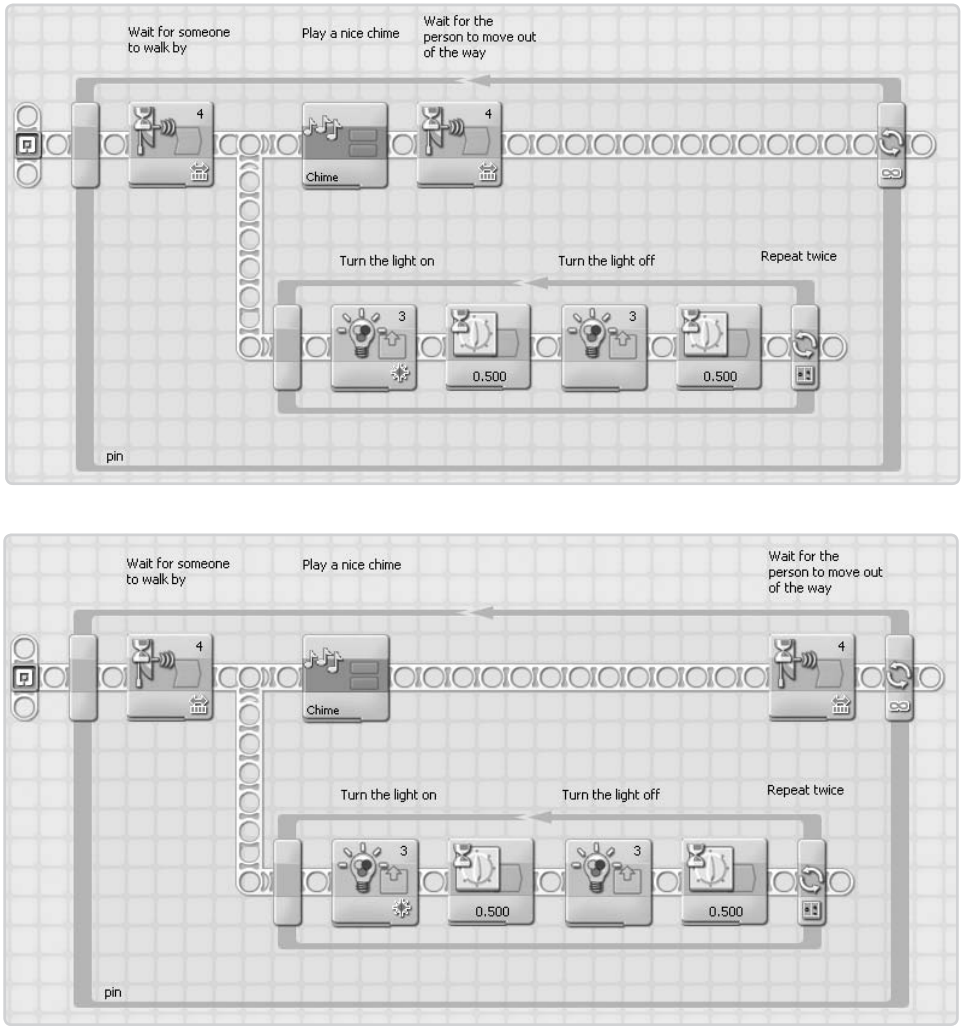


Figure 17-8: The completed DoorChime program

understanding program flow rules

Using multiple Sequence Beams complicates the rules concerning program flow in several ways. For example, a program ends only once it reaches the end of all the Sequence Beams. In this section, I'll discuss the other program flow rules that are affected by using multiple Sequence Beams.

starting blocks and data wires

A block can start running only after there are values on all the data wires attached to the block, as demonstrated by the BlockStartTest program in Figure 17-9. The Display block on the top Sequence Beam displays 1 and the Text block writes 2 to the data wire (to be used by the other Display block). The Display block on the bottom Sequence Beam can't start running until after the Text block puts a value on the data wire. When you run this program, the display will show 1, and then after a one-second pause, the display will show 2.

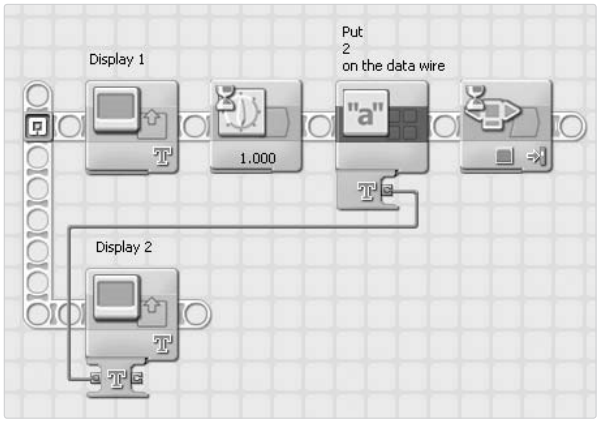


Figure 17-9: The BlockStartTest program

starting a loop or switch block

A Loop or Switch block can't start until there are values on all the data wires that enter the block. The LoopStartTest program shown in Figure 17-10 demonstrates this rule. The Loop block can't start until after the Text block puts a value on the data wire, even though this value isn't used until the program gets to the second Display block within the Loop block. When you run this program, it will display 1 (from the Display block on the top beam) and then pause for one second. After the Text block puts a value on the data wire, the Loop block will start, and the program will display 2 and then wait for an additional second before displaying 3. This example uses a Loop block, but the same rule applies to a Switch block.

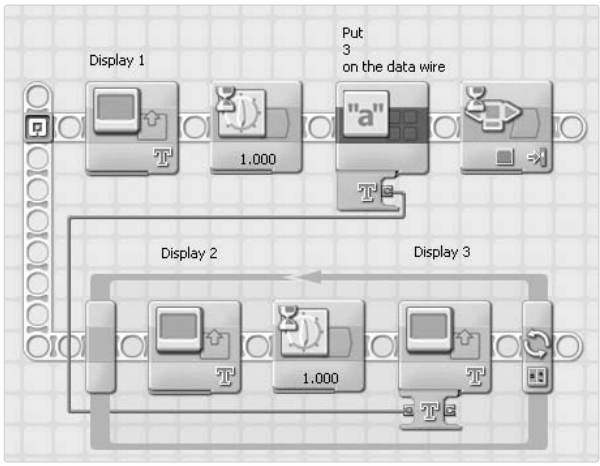


Figure 17-10: The LoopStartTest program

using values from a loop or switch block

A data wire that starts inside a Loop block and connects to a block outside the Loop block will have a value only when the Loop block finishes. The LoopCountTest program shown in Figure 17-11 shows an example of this rule. The Loop block repeats five times, pausing for a total of five seconds. The DisplayNumber block on the bottom Sequence Beam displays the value written to the data wire from the Loop block's Loop Count data plug. Because the DisplayNumber block is outside the Loop block, only the last value (4) is passed to it on the data wire, and only after the Loop block has finished. When you run this program, it will pause for five seconds and then display 4. This rule also applies to the Switch block; data wires that leave a Switch block will have a value only after the Switch block completes.

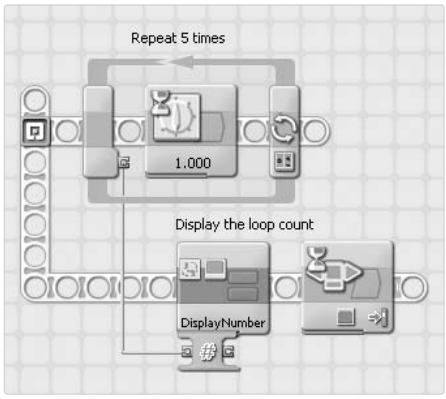


Figure 17-11: The LoopCountTest program

using my blocks

Only one copy of a particular My Block can run at the same time, as demonstrated by The MyBlockTest program in Figure 17-12. The ProgTimer3 My Block is a programmable timer based on the Timer3 program developed in Chapter 10. When you run this program, it will display 1, pause for one second, display 2, pause for another second, and then display 3. The second pause happens because the ProgTimer3 block on the top Sequence Beam won't start until the ProgTimer3 block on the bottom Sequence Beam completes.

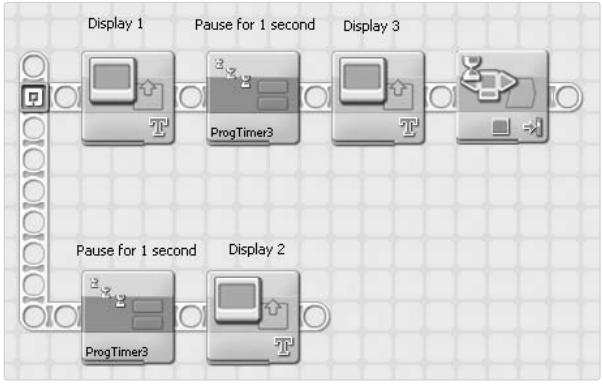


Figure 17-12: The MyBlockTest program

This rule applies only to My Blocks, so if you replace the two ProgTimer3 blocks with Wait Time blocks, the two pauses will happen simultaneously. The program will display 1, pause for one second, display 2, and then almost immediately display 3 (so quickly that you won't see the 2). This rule also only applies to two copies of the same My Block. Two ProgrTimer3 blocks won't run at the same time, but a ProgTimer3 block and a DisplayNumber block will.

NOTE **If you really need two copies of the same My Block to run in parallel, open the My Block, and select the File ▸ Save As menu item to save a copy of the My Block using a different name. Then you'll have two different My Blocks that do the same thing and will be able to run at the same time.**

This behavior is really only apparent with My Blocks that wait for something to happen. For example, you won't notice if two DisplayNumber blocks don't run simultaneously. Most My Blocks start and finish quickly enough that this rule doesn't have an appreciable effect on most programs.

synchronizing two sequence beams

You can control when a second Sequence Beam starts by choosing where to start drawing the Sequence Beam. In some programs, you may want to also make the task on one Sequence Beam pause until the task on the other Sequence Beam completes. The following are two ways to accomplish this.

the AroundTheBlock program

In the AroundTheBlock program (see Figure 17-2), you may want to stop updating the display after the TriBot has completed moving around the square. Instead of having the Loop block run forever, you can use a variable to control when the loop stops. Figure 17-13 shows a modified version of the AroundTheBlock program that uses a logic variable named *Done* to control the Loop block. The variable is set to false before the TriBot starts moving and is set to true after all the movement is complete. The Loop block on the bottom Sequence Beam will continue displaying the motor position until the variable is set to true by the Variable block on the top Sequence Beam.

the DoorChime program

In the DoorChime program (see Figure 17-8), I moved the Ultrasonic Sensor block to the right side of the Loop block to show that it shouldn't start until after the light stops flashing. Arranging the blocks this way makes the intent of the program easier to understand, but it doesn't force the program to work correctly. The program could stop working if you change the timing of either the Chime block or the blocks

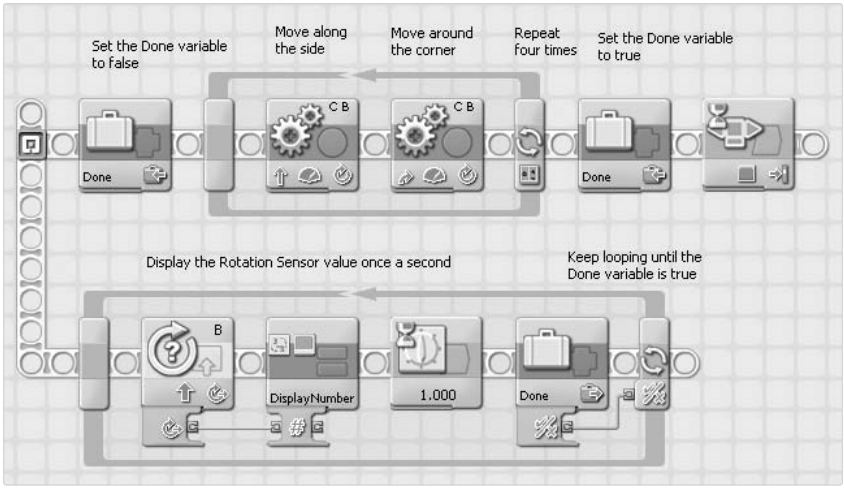


Figure 17-13: Synchronizing the Sequence Beams in the AroundTheBlock program

that flash the light, perhaps by adding a few more notes to the Chime block or making the light flash faster.

The changes to the DoorChime program shown in Figure 17-14 make the blocks on the top Sequence Beam wait until the blocks on the bottom Sequence Beam are finished. The synchronization is provided by the two Math blocks and the data wire connecting them. The settings of these two blocks are unimportant; all that matters is that the Math block on the top Sequence Beam can't start until after the one on the bottom Sequence Beam completes. Adding these two blocks and the data wire between them ensures that the Ultrasonic Sensor block won't run until after the chime is complete and the light has finished flashing.

Using data wires in this way allows you to synchronize the Sequence Beams by adding only two blocks and a data wire. However, the resulting program can be difficult to understand, especially if you neglect to add a comment that makes it very clear that the two Math blocks are there only for synchronization and are not really doing any math. This approach also depends on the special behavior of data wires and Sequence Beams, and as a general principle, it's usually better to avoid programming special cases like this.

The solution used in the AroundTheBlock program (using a variable to indicate when a task is done and checking the variable in a loop) requires more blocks but results in a much clearer program. Both methods are widely used in

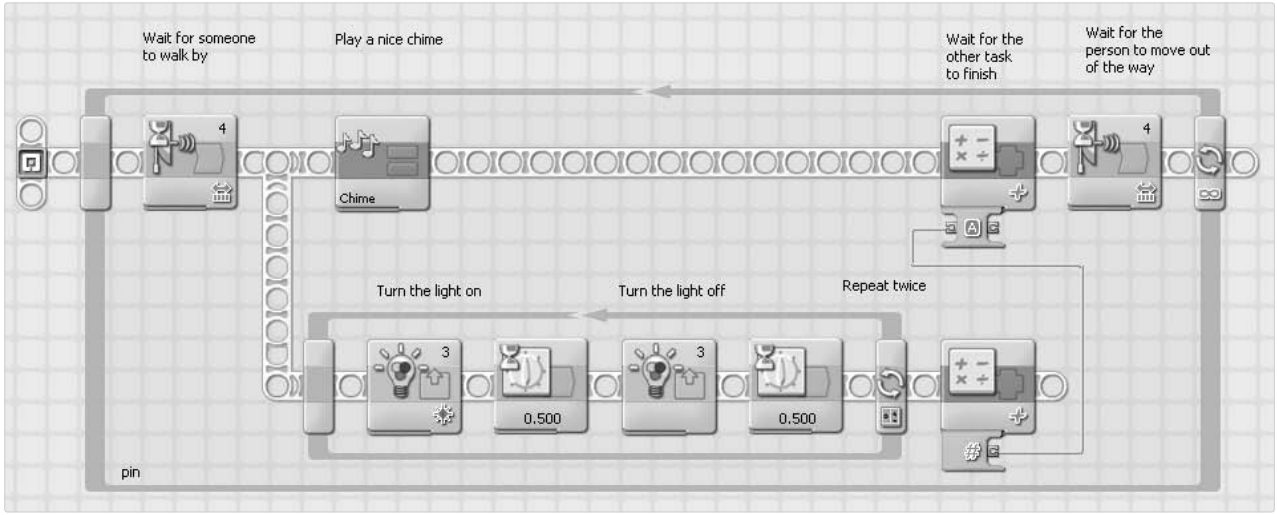


Figure 17-14: Synchronizing the Sequence Beams in the DoorChime program

programs and examples you'll find on popular NXT websites; the approach you choose to use in your programs is mostly a matter of personal preference.

keeping out of trouble

Adding a second Sequence Beam lets your program do more but also increases the number of ways that things can go wrong. Using multiple Sequence Beams affects almost every aspect of NXT-G programming, including variables, data wires, My Blocks, and program flow. This is an advanced programming technique that can allow you to write some incredible programs but can also cause a lot of confusion. Here are some tips to help you avoid the most common problems:

- **Use a second Sequence Beam only when it's really necessary.** If you can find a solution to your problem that requires only one Sequence Beam, then you're usually better off avoiding the added complexity.
- **Go slow when editing the program.** The IDE tends to get confused much easier when editing programs with two or more Sequence Beams.
- **Avoid trying to control the same motor or sensor from more than one Sequence Beam.**
- **Use variables instead of data wires to pass information between Sequence Beams.** This often makes your program easier to understand.
- **Be especially careful with data wires that pass into or out of Loop blocks and Switch blocks.** (Reread "Understanding Program Flow Rules" on page 229 if you're not sure what to watch out for.)

conclusion

Using multiple Sequence Beams allows your program to perform more than one task simultaneously, a form of multitasking. The changes made in this chapter to the AroundTheBlock and DoorChime programs demonstrate two simple ways to enhance a program by adding a second Sequence Beam.

Although multitasking is a very useful programming technique, it comes at the cost of added complexity. The simple program flow rules that you're familiar with become more complicated when using more than one Sequence Beam. For this reason, multitasking works best with small, independent tasks.

18

the LineFollower program

Now that you've seen the advanced programming features that NXT-G has to offer, it's time to put them to use. In this chapter, I'll walk you through the process of transforming the simple line-following program from Chapter 6 into a much more flexible and accurate program.

The chapter begins with a brief discussion on line following. Then I'll show you how to use files and a configuration program to collect and save the settings used by the LineFollower program. In the second half of the chapter, I'll show you how to use a more complicated control strategy for following a line. By the end of the chapter, your TriBot will be able to follow a line quickly and accurately.

following a line

Programming your robot to follow a line is an interesting challenge because you can make your program as simple or as complicated as you want. It's fairly easy to write a simple program like the ones presented in Chapter 6 that allows the TriBot to slowly follow a line with wide turns. By writing a more complex program, you can make the robot move faster, handle tighter turns, and even detect and avoid obstacles.

requirements

Let's begin by thinking about the general requirements of a line-following robot. The simplest requirement is that the robot must follow the line; it shouldn't wander off the line and get lost to one side or the other. You'll usually want the robot to move as quickly as possible, if for no other reason than that it's just more fun to watch a fast-moving robot. For the programs presented in this chapter, speed is more of a goal than a hard and fast requirement, though in another setting, such as a competition, you might have a fixed-speed requirement. In addition, the program should be flexible enough to work with different lines, lighting conditions, and sensors. It can be quite annoying when your robot (and its program) works great when you're building it but fails miserably when you bring it to your friend's house. These requirements can be summarized as follows:

* Stay on the line.
* Move as quickly as possible.
* Be flexible.

assumptions

To keep the program simple, I'll make the following assumptions:

* The robot will start next to the line so it doesn't have to hunt for the line.
* The line will be at least as wide as the lines on the LEGO test pads (about $\frac{3}{4}$ inch or 2 cm). If the line is too thin, it's easy for the robot to accidentally cross it.
* The line won't have corners or very sharp curves.

- The line won't cross over itself. A figure-eight course will tend to confuse most line followers.
- The line will be dark and the background will be light to simplify the discussion of the programs.

Even with these assumptions, the program will be fairly complicated by the end of the chapter. Of course, you can expand the final program to address any of these issues on your own.

the starting point

As a starting point for this discussion, I'll use the Line-Follower program from Chapter 6, as shown in Figure 18-1. This program allows the TriBot to follow a line, but it works only if the robot moves slowly.

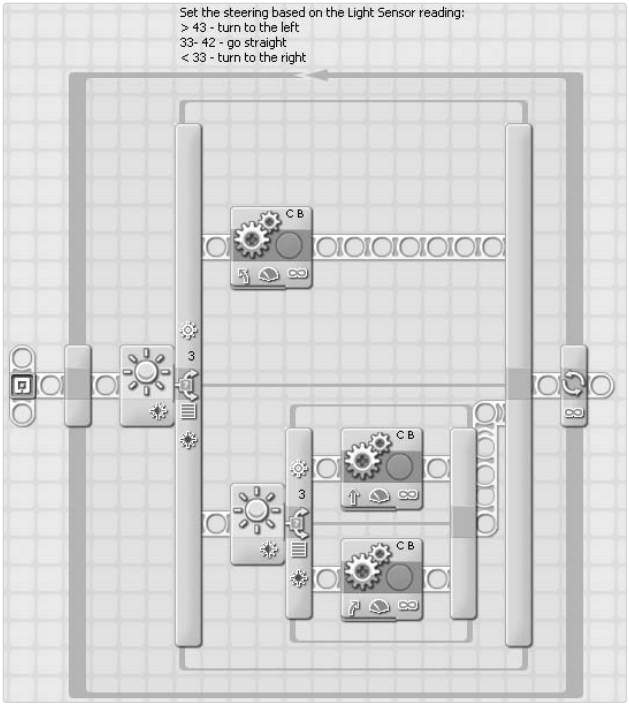


Figure 18-1: The original LineFollower program

Although the Light Sensor is used in Figure 18-1 and in the images in this chapter, the program will work just as well with the Color Sensor. As we proceed, I'll give instructions for using either sensor.

To follow a line, the Light Sensor (or Color Sensor) should be mounted on the front of the TriBot pointing downward, as shown in Figure 18-2.

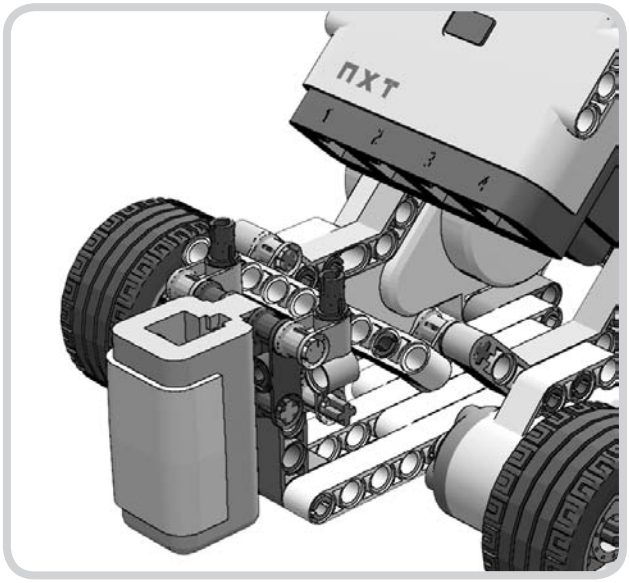


Figure 18-2: Mounting the sensor on the front of the TriBot

selecting the sensor trigger values

The first enhancement will address the flexibility requirement. The LineFollower program shown in Figure 18-1 uses *hard-coded* values, meaning the trigger values for the Light Sensor are set in the program and can't easily be changed. The settings work fine with my NXT using the LEGO test pad, with the lighting in my office, but they may not work as well with a different Light Sensor, line, or different lighting. To avoid hard-coding the trigger values, you can write a program that collects the information needed to determine the trigger values. This makes the line-following program more flexible because it can adapt to different sensors, lines, and lighting conditions.

Since you're using a dark line on a light background, the value from the Light Sensor should be smallest when

the sensor is directly over the line and largest when it's completely off the line and over the background. Using these two values (the smallest and largest Light Sensor readings) and a little math, you can calculate the trigger values needed for the LineFollower program.

In this section, you'll first write the LineFollowerConfig program to collect two readings from the Light Sensor and write them to a file. Then you'll modify the LineFollower program to read the values from the file and use them to calculate the trigger values.

building the LineFollowerConfig program

The LineFollowerConfig program's job is to read two values from the Light Sensor and save them to a file. The first value is read when the TriBot is over the center of the line, and the second is read when the TriBot is completely off the line. One key is to place the TriBot correctly (either on or off the line) before reading the Light Sensor. A simple approach is to use the NXT's screen to give instructions on where to put the robot. For example, you can tell the person using the program to place the TriBot over the line and then wait for the user to press a button before reading the sensor. You can then repeat the process to get the second sensor reading with the robot off the line.

To save the Light Sensor readings to a file, simply connect a Light Sensor block to a File Access block. I'll use *LF_Config* for the filename (*LineFollower_Config* is more descriptive but is longer than the 15-character limit on the filename). Recall that writing to a file always adds new information to the end of the file. Since you want to store new values each time you run the program, the first thing the program should do is delete the file if it already exists. Also, remember to close the file after writing the two values.

Now that you know what the program should do, you can use pseudocode to describe the program, as shown in Listing 18-1.

```
delete the LF_Config file
use Display blocks to tell the user to place the
   robot over the line
wait for the Enter button to be bumped
read the Light Sensor
write the value to the LF_Config file
use Display blocks to tell the user to place the
   robot off the line
wait for the Enter button to be bumped
read the Light Sensor
write the value to the LF_Config file
close the LF_Config file
```

Listing 18-1: The FileFollowerConfig program

Figure 18-3 shows the first part of the program, which deletes the file and then collects the Light Sensor reading when the TriBot is placed over the line.

The first File Access block deletes the file if it already exists. If you forget this step, then each time you run the program, it will add the two readings onto the end of the file, instead of replacing the previous values.

You'll need to enter the name of the file in the Configuration Panel for this block (shown in Figure 18-4). You can then copy this block (hold down the CTRL key while dragging the block) to create the other File Access blocks in the program. This is an easy way make sure that none of the File Access blocks accidentally uses the wrong filename.

Figure 18-4: Deleting the old file

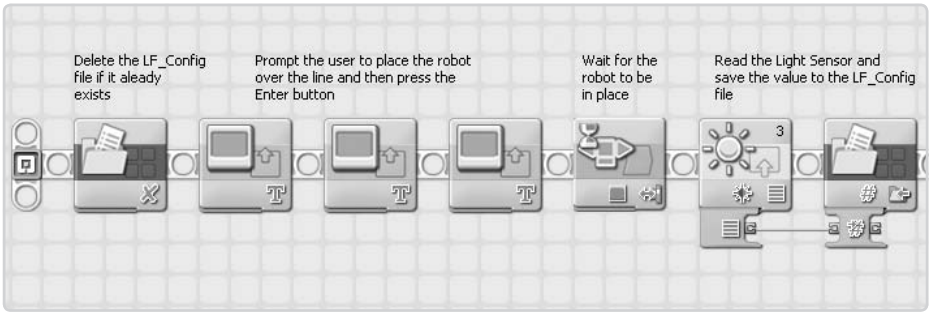


Figure 18-3: Collecting the first Light Sensor reading

Figures 18-5 through 18-7 show the Configuration Panels for the three Display blocks that give the instructions. Each Display block prints one line of instructions, and only the first block clears the display.

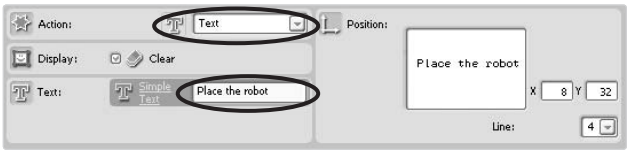


Figure 18-5: Displaying the first line of the instructions

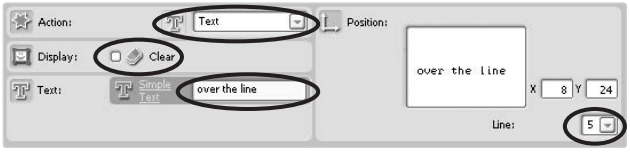


Figure 18-6: Displaying the second line of the instructions

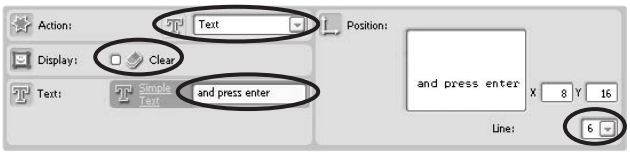


Figure 18-7: Displaying the third line of the instructions

After displaying the instructions, the program waits for you to bump the Enter button, which you'll do after placing the TriBot in the correct spot. Figure 18-8 shows the Configuration Panel for the Wait block.

Figure 18-8: Waiting for the Enter button to be bumped

The Light Sensor block (shown in Figure 18-9) uses all the default settings. For the Color Sensor, use the settings shown in Figure 18-10.

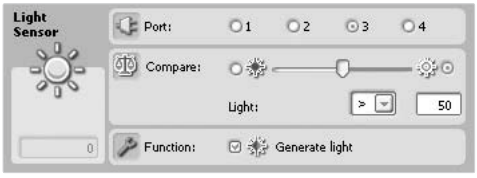


Figure 18-9: Reading the Light Sensor

Figure 18-10: Reading the Color Sensor

The File Access block (shown in Figure 18-11) writes the value from the Light Sensor or Color Sensor block to the *LF_Config* file. Be sure to draw a data wire from the Light Sensor block's Intensity data plug to the File Access block's Number data plug, as shown in Figure 18-3. For the Color Sensor block, use the data plug labeled *Detected Color*, which is used for the intensity when the sensor is in Light Sensor mode.

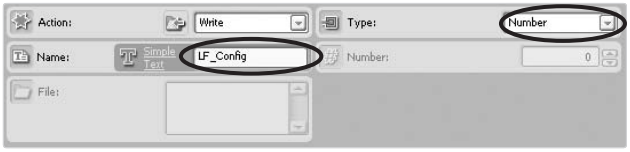


Figure 18-11: Writing the sensor reading to the file

The second half of the program uses a similar group of blocks to collect the Light Sensor reading with the TriBot off the line and then closes the file. Figure 18-12 shows the entire program, in two pieces.

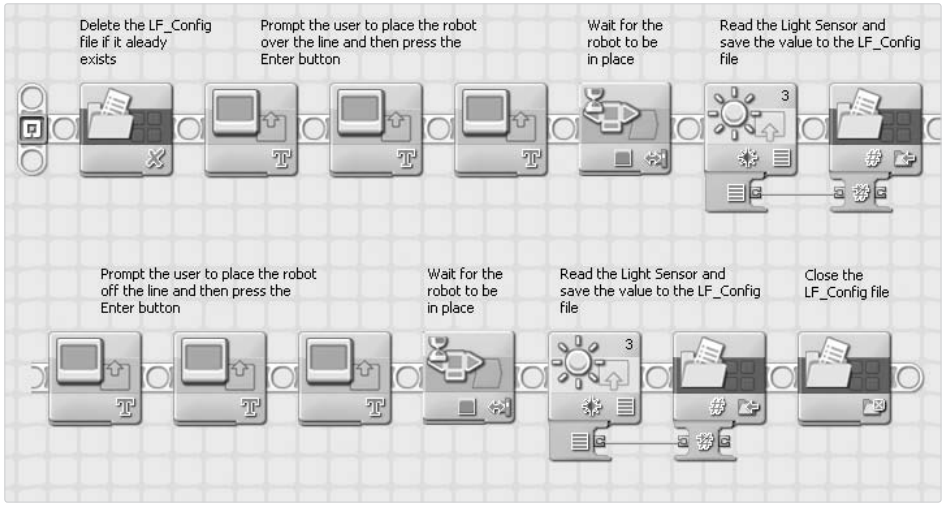


Figure 18-12: Collecting the two Light Sensor readings

The settings for the blocks used to display the instructions and save the sensor reading are almost identical to the settings used in the first half of the program. The only block that needs to change is the second Display block, which instructs you to place the TriBot off the line, as shown in Figure 18-13.

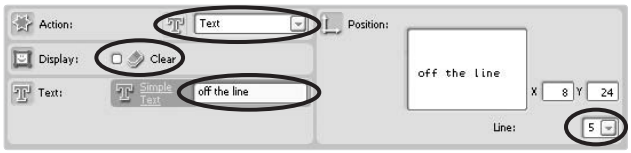


Figure 18-13: Placing the robot off the line

The final File Access block, shown in Figure 18-14, closes the file.

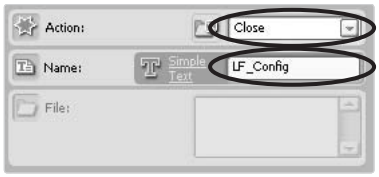


Figure 18-14: Closing the file

testing the LineFollowerConfig program

Now download and run the program. It should display the first set of instructions and then wait for you to press the Enter button before turning on the small light on the Light Sensor. After you press Enter, the program should display the second set of instructions and again wait for you to press Enter. The program ends after you press Enter the second time.

NOTE **If the program ends after you press Enter the first time, the Action setting on the Wait block is probably wrong. If the Action setting is Pressed instead of Bumped, the program will quickly run all the remaining blocks in the program while you're pressing Enter.**

It's easy to tell whether the program goes through all the steps by watching the display, but how do you tell whether reasonable values are being written to the file? You can easily modify the FileReader program from Chapter 15 (shown in Figure 18-15) to display the contents of the *LF_Config* file by setting the name in the File Access block's Configuration Panel to *LF_Config*, as shown in Figure 18-16.

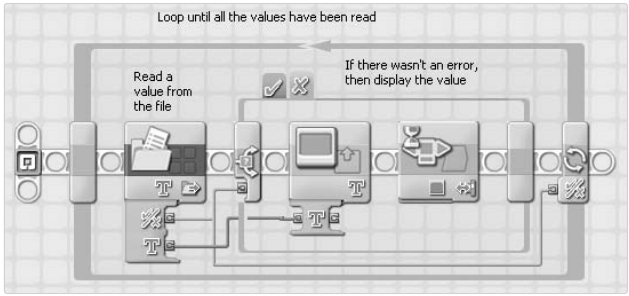


Figure 18-15: The FileReader program

Figure 18-16: Selecting the LF_Config *file*

When you run the FileReader program, it should display the two values collected by the LineFollowerConfig program. Both values should be greater than zero, and the first value should be less than the second one. For example, the values from my test run are 22 and 47. If you get zero for one (or both) of the values, it usually means that the sensor is not connected to the correct port or that there is a problem with the data wire connection between the Light Sensor block and the File Access block. If the first value is larger than the second one, then you may have the instructions reversed, or you may not be following the given instructions.

changing the LineFollower program

In this section, you'll modify the LineFollower program to use the values collected by the LineFollowerConfig program. The changes will be made in three parts: reading the values from the *LF_Config* file, calculating the trigger values for the two Switch blocks, and then using the trigger values to control the Switch blocks.

reading the high and low values

The first set of changes involves reading the two values from the *LF_Config* file so they can be used by the rest of the program. The LineFollowerConfig program first writes the lower of the two values (the one taken while the robot is over the line), followed by the higher value (the one taken while the robot is off the line). The LineFollower program must read the two values in the same order; the lower one first and then the higher one. The changes to the program, shown in Figure 18-17, use two number variables, High and Low, to hold the values read by the File Access blocks. The third File Access block closes the file after the two values have been read.

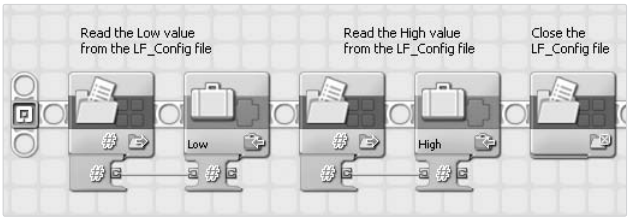


Figure 18-17: Reading the High and Low values from the LF_Config *file*

Figure 18-18 shows the Edit Variables window with the Low and High variables defined.

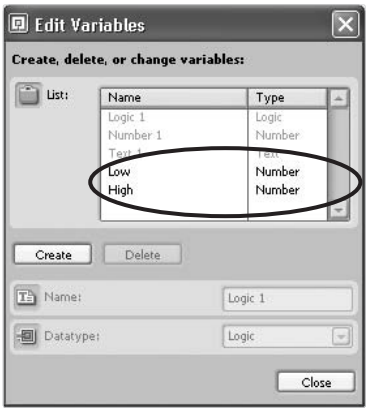


Figure 18-18: Defining the Low and High variables

The first two File Access blocks use the same settings (shown in Figure 18-19) to read a number from the *LF_Config* file. Setting the filename is easier if you first connect your NXT to the MINDSTORMS software so that you can select the name from the list. Because NXT-G is a graphical language, there aren't many places where you can make a typographical error, but entering the name of a file is one of them.

Figure 18-19: Reading a number from the LF_Config *file*

Figures 18-20 and 18-21 show the Configuration Panels for the two Variable blocks that store the values read by the File Access blocks. The first Variable block writes to the Low variable, and the second one writes to the High variable.

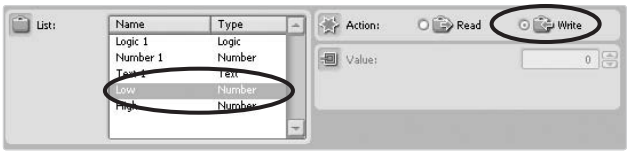


Figure 18-20: Storing the Low value

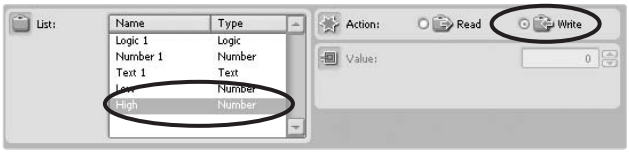


Figure 18-21: Storing the High value

Figure 18-22 shows the Configuration Panel for the final File Access block, which closes the file after the program is through using it.

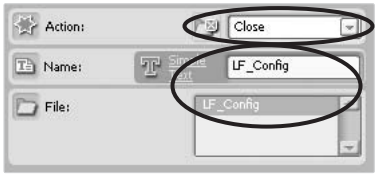


Figure 18-22: Closing the LF_Config

Once this section of the program has completed, the Low and High variables should contain the two values collected by the LineFollowerConfig program. The next section of code will use these values to calculate the trigger values the program will use.

calculating the trigger values

The original program uses two trigger values, one for each of the Switch blocks shown in Figure 18-1. When the Light Sensor reading is greater than the higher of the two trigger values, the TriBot steers to the left; when the reading is less than the lower trigger value, the TriBot steers to the right; and when the reading is between the two trigger values, the TriBot moves straight. In Chapter 6, I determined the trigger values by finding the Light Sensor reading when the robot was placed over the edge of the line. Then I added 5 to get the higher trigger value and subtracted 5 to get the lower trigger value. I'll take a similar approach here.

When the Light Sensor is over the edge of the line, the value it reports should be about halfway between the low and high values read from the *LF_Config* file. You can calculate this value using the following formula:

Target = (High + Low) ÷ 2

This is called the *target* value because this is the value the Light Sensor should read when the robot is in just the right position over the edge of the line. To determine the trigger values for the two Switch blocks, which I'll call *HighTrigger* and *LowTrigger*, add and subtract 5 from the target value:

HighTrigger = Target + 5
LowTrigger = Target – 5

Performing these calculation in NXT-G is a simple matter of stringing together some Math and Variable blocks. Figure 18-23 shows the blocks used to compute the target value and store it in the Target variable. (This new code should be placed just after the File Access block that closes the *LF_Config* file.)

Figure 18-24 shows the Edit Variables window with the three new variables defined.

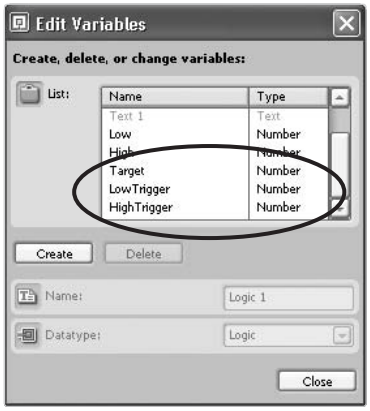


Figure 18-24: Defining the Target, LowTrigger, and HighTrigger variables

The two variable blocks, shown in Figures 18-25 and 18-26, read the values from the Low and High variables.

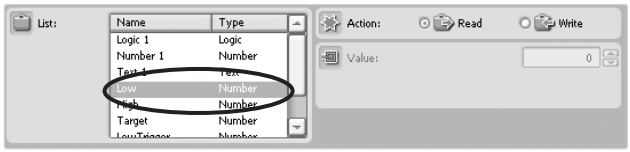


Figure 18-25: Reading the Low variable

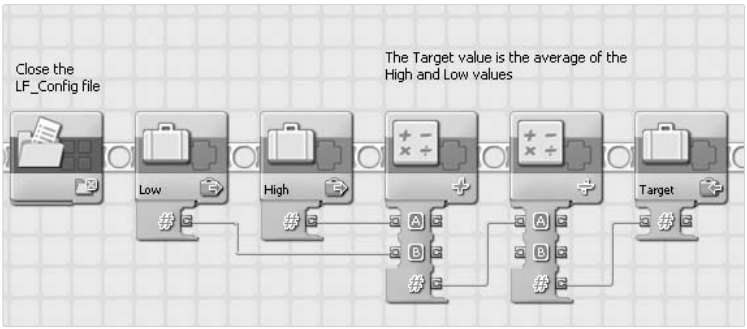


Figure 18-23: Calculating the target *value*

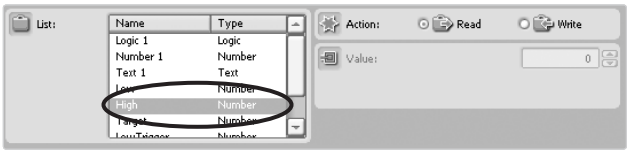


Figure 18-26: Reading the High variable

Figures 18-27 and 18-28 show the Configuration Panels for the two Math blocks, which first add the High and Low values together and then divide the sum by 2 to get the average, or middle, value.

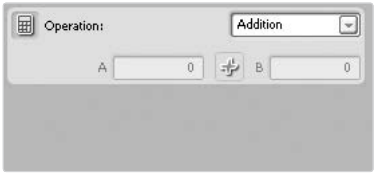


Figure 18-27: Adding the Low and High values

Figure 18-28: Dividing the sum by 2

Figure 18-29 shows the Configuration Panel for the final Variable block, which stores the result from the Math block in the Target variable.

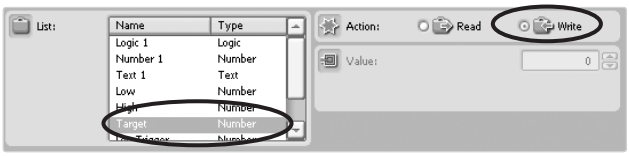


Figure 18-29: Storing the target *value*

Once you have the target value, you can calculate the HighTrigger and LowTrigger values by adding and subtracting 5 using the code shown in Figure 18-30.

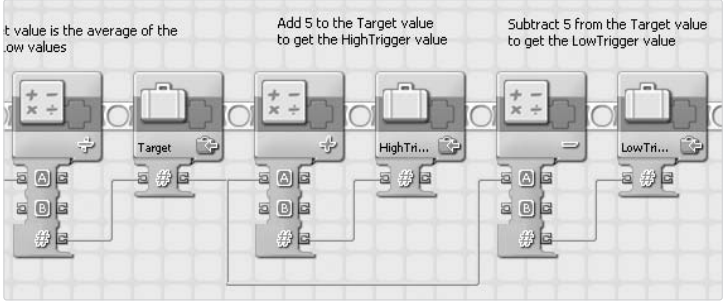


Figure 18-30: Calculating the LowTrigger and HighTrigger values

Figure 18-31 shows the Configuration Panel for the first Math block, which adds 5 to the target value. Figure 18-32 shows the Configuration Panel for the Variable block, which stores the resulting value in the HighTrigger variable.

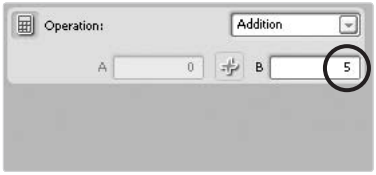


Figure 18-31: Adding 5 to the target *value*

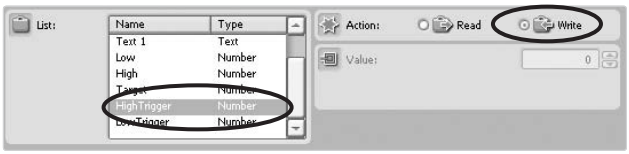


Figure 18-32: Storing the HighTrigger value

The second pair of blocks, shown in Figures 18-33 and 18-34, subtracts 5 from the target value and stores the result in the LowTrigger variable.

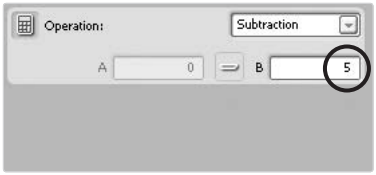


Figure 18-33: Subtracting 5 from the target *value*

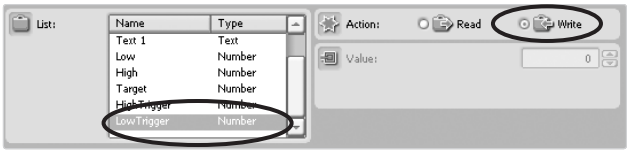


Figure 18-34: Storing the result in the LowTrigger variable

When the program reaches the end of the code shown in Figure 18-30, the trigger values for the two Switch blocks should be stored in the HighTrigger and LowTrigger variables. The next step is to change the program to use these two values.

using the trigger values

In the original program (shown in Figure 18-1), the two Switch blocks read the value from the Light Sensor, compare it to the hard-coded trigger values, and then decide which group of blocks to run. Unfortunately, the Switch block doesn't have a data plug for supplying a trigger value. Since you don't know what the trigger value will be until the program runs, you can't just enter the value into the Switch block. To use a calculated trigger value, you need to use a Light Sensor block to read the sensor and compare the value to the trigger value. The Light Sensor block can then supply the result of the comparison to the Switch block, which can then decide which group of blocks to run.

Figure 18-35 shows the changes to the main part of the LineFollow program. Each time through the loop, the Light Sensor reading is first compared with the HighTrigger value. If the sensor reading is greater than the trigger, then the Switch block will run the Move block on the upper Sequence Beam, which will steer the TriBot to the left. If the sensor reading isn't greater than the HighTrigger value, the reading from the Light Sensor is compared with the LowTrigger value. If the sensor reading is greater than the trigger value, the Switch block will run the Move block on the upper Sequence Beam, which moves the TriBot in a straight line. If the sensor reading is at or below the trigger value, the Switch block will run the Move block on the lower Sequence Beam, which will steer the TriBot to the right.

Figures 18-36 and 18-37 show the Configuration Panels for the two Variable blocks that read the HighTrigger and LowTrigger values. The values are supplied to the Light Sensor blocks by connecting the Variable block's Value data plug to the Light Sensor block's Trigger data plug.

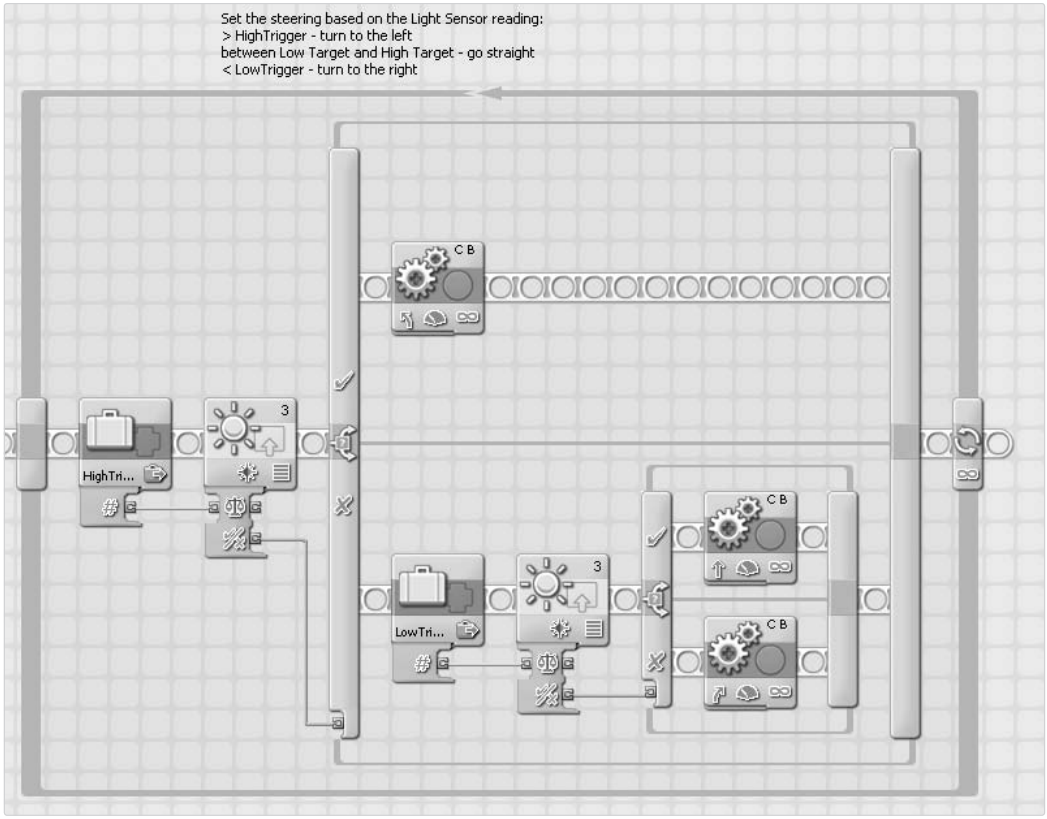


Figure 18-35: Comparing the Light Sensor with the trigger values to steer the robot

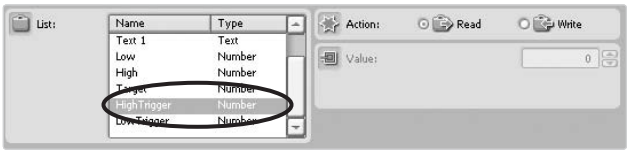


Figure 18-36: Reading the HighTrigger variable

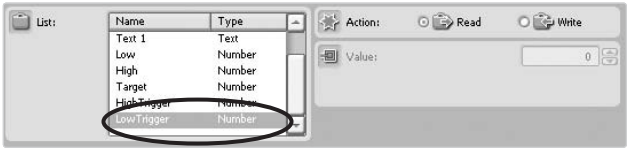


Figure 18-37: Reading the LowTrigger variable

The two Light Sensor blocks use all the default configuration settings, as shown in Figure 18-38. Both use identical settings because the only difference between them is the trigger values they use, and those values are supplied using a data wire. If you're using the Color Sensor instead of the Light Sensor, use the Color Sensor block with the settings shown in Figure 18-39.

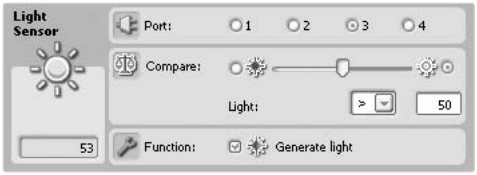


Figure 18-38: The Configuration Panel for the Light Sensor block

Figure 18-39: The Configuration Panel for the Color Sensor block

The two Switch blocks need to be changed to use the value from the Light Sensor block's Yes/No data plug, which reports the result of comparing the sensor reading with the trigger value. Figure 18-40 shows the Configuration Panel for the Switch blocks. (Both Switch blocks use the same settings.)

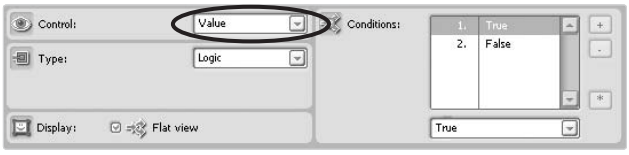


Figure 18-40: The Configuration Panel for the Switch blocks

testing the LineFollower program

With all the changes made to the program, you're now ready for testing. Be sure to first run the LineFollowerConfig program to collect the two Light Sensor values, and then place the TriBot on the right edge of the line and start the LineFollow program. The TriBot should be able to follow a line at least as well as the original program and should adapt better to different lines, sensors, or lighting conditions.

If the program doesn't work, the first thing to do is use the FileReader program to check the contents of *LF_Config*. I usually see a low value of around 25 and a high value around 50. If the values are 0 or if the first value is larger than the second, check the LineFollowerConfig program. If the values in *LF_Config* are correct but the program still doesn't work, check the settings on the Math blocks and the data wire connections. You can also add some DisplayNumber blocks just before the Loop block to show the values of the variables. The exact values of the variables will depend on your particular conditions, but the following statements about the values should be true:

- The Low value should be smaller than the High value.
- The Target value should be the average of the High and Low values.
- The HighTrigger value should be 5 more than the Target value.
- The LowTrigger value should be 5 less than the Target value.

By examining the variables, you should be able to tell which Math block or data wire connections to check. If all the variable values are reasonable, then the problem most likely involves either one of the Light Sensor or Switch blocks.

adding a pause at the beginning of the program

While testing the program, you may find that the robot starts off a little too quickly and runs into your hand. Fortunately, you can easily solve this problem by using a Wait Time block to add a small pause to the start of the program, as shown in Figure 18-41. Figure 18-42 shows the Configuration Panel for the Wait Time block.

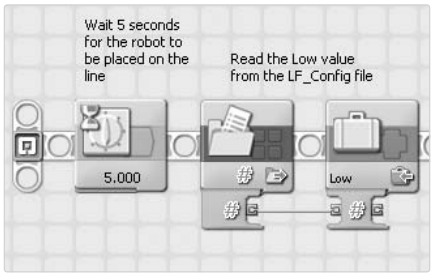


Figure 18-41: Waiting for you to move your hand

Figure 18-42: The Configuration Panel for the Wait Time block

improving the control algorithm

The part of a line-follower program that adjusts the steering based on the Light Sensor reading is called a *control algorithm*. Improving the control algorithm will allow the TriBot to move faster and follow lines with tighter turns.

The control algorithm you've been using so far is called a *three-state controller*, because the program will do one of three things based on the light sensor reading: go straight, turn left, or turn right. This approach works as long as the TriBot moves slowly and the line doesn't curve too quickly. The main problem with this method is that when the robot needs to turn, it always turns the same amount, because the Steering values for the Move blocks are hard-coded. If the robot encounters a sharp or gentle turn, it will still turn at

WHY USE VARIABLES?

The LineFollower program can be written without variables if you replace the Variable blocks with data wires to move the values through the program. For example, Figure 18-43 shows the program with the Variable blocks removed.

Removing the variables gives you a smaller program, which means you can see more of it on your computer screen, and it will take up slightly less memory when you download it to the NXT. The drawback of removing the variables is that the program can become a little more difficult to understand and harder to debug.

I find it helpful to use variables when first developing a program. That way, I can look at the values a program is using by just adding a Variable block followed by a DisplayNumber block. Once the program is working, it's easy to remove the Variable blocks if you want to make the program a little shorter.

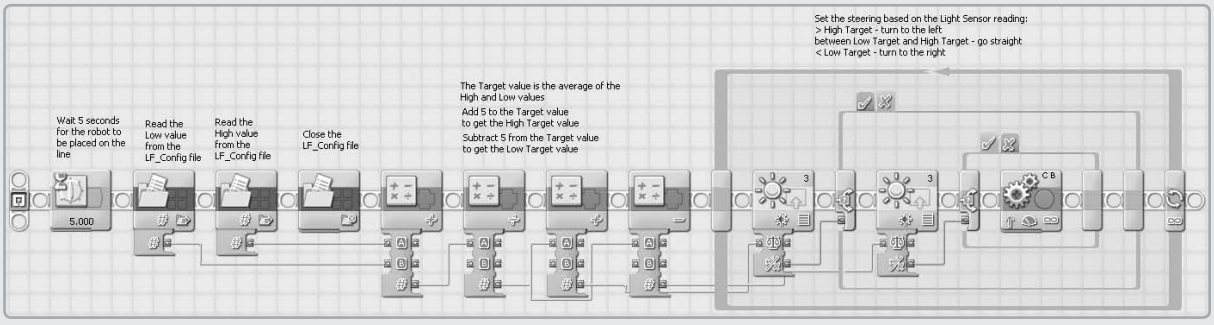


Figure 18-43: Removing the Variable blocks

the same steering value. It would be nice if the steering value was dependent on the sharpness of the line, with little steering for straight lines and sharp steering for sharp corners. This would allow the program to respond quickly to changes in the direction of the line while still moving smoothly when the line is straight.

You can implement this feature by making the robot turn just a little when it's close to the edge of the line and turn a lot when it's far from the edge of the line. This approach is called a *proportional controller*, because the change made to the steering is proportional to, or directly related to, the robot's distance from the edge of the line. There are three steps to using a proportional controller to follow a line:

1. Determine how far the TriBot is from the edge of the line.
2. Decide how to control the motors.
3. Decide how much the TriBot should steer left or right based on its distance from the edge of the line.

In the following sections, you'll build the LineFollower2 program. There are enough changes from the LineFollower program that it will be easier to start almost from scratch rather than modify the existing program. You can reuse the code at the very beginning of the LineFollower program to read *LF_Config* and compute the target value (shown in Figures 18-44 and 18-45) as a starting point for the LineFollower2 program.

how far from the edge?

Once the blocks in Figures 18-44 and 18-45 have run, the Target variable will hold the value that the Light Sensor should read when the TriBot is over the edge of the line. You can determine how far the TriBot is from the edge of the line by comparing the reading from the Light Sensor with the target value. As the TriBot moves farther from the edge of the line, the difference between the Light Sensor reading and the target value will increase, and when the TriBot is directly over the edge of the line, the Light Sensor reading should exactly match the target value.

understanding the error value

The difference between the sensor reading and the target value is called the *error value*. You can think of the error value as the difference between where you want the robot to be and where the robot actually is. The error value is computed using this simple formula:

Error = Target – Sensor Reading

For example, say the *LF_Config* file contains 22 for the low value and 48 for the high value. The target value will be (48 + 22) ÷ 2, which is 35. Table 18-1 shows the error values for a range of Light Sensor readings, using 35 as the target value. You can see from this table that the error value increases as the Light Sensor reading gets farther from the target value. Also notice that the error value is positive when the sensor reading is less than the target value, and it's negative when it's greater than the target value. The sign of the error value (positive or negative) will determine whether the TriBot should turn left or right.

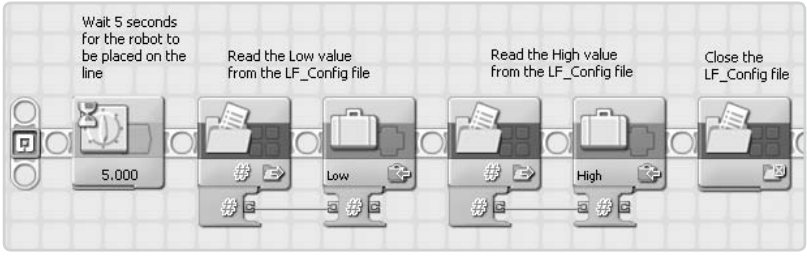


Figure 18-44: Reading the Low and High values from the LF_Config *file*

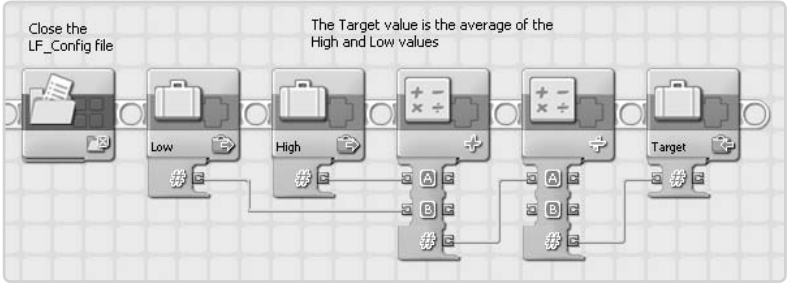


Figure 18-45: Calculating the target value

table 18-1: error values for various light sensor readings

light sensor reading	error value *target – light sensor reading*
22	13
27	8
33	2
35	0
37	-2
43	-8
48	-13

using percent error

You can make one improvement to how the error value is computed. The error values shown in Table 18-1 depend only on the target value, not the low and high values, which means that you would get the same error values if the low and high values are 0 and 70, are 30 and 40, or are any pair of numbers whose average is 35. However, the error value should reflect the position of the TriBot, which will be somewhere between the center of the line and completely off the line. The error values in Table 18-1 give you some useful information but don't really tell you the whole story about how far the robot is from the edge. For example, if the Light Sensor reading is 30, the error value is 5, and if the low value is 30, this error value means that the TriBot is completely off the line. But if the low value is 0, then an error value of 5 means the TriBot is only slightly to the right of the line.

The error makes more sense if you look at it as a percentage of the possible range of values (from the low to high value). To calculate the range, subtract the low value from the high value:

Range = High – Low

Now you can get the error value as a percent of the range using this formula:

Percent Error = (Target – Sensor Reading) × 100 ÷ Range

Multiplying by 100 before performing the division allows the formula to work using either the integer math in NXT-G 1.1 or the floating-point math in NXT-G 2.0. Table 18-2 adds the percent error to the values in Table 18-1, using 35 for the target value and 26 for the range.

table 18-2: percent error values for various light sensor readings

light sensor reading	error value *target – light sensor reading*	percent error value *(target – sensor reading) × 100 ÷ range*
22	13	50
27	8	30
33	2	7
35	0	0
37	-2	-7
43	-8	-30
48	-13	-50

Returning to the earlier example, an error value of 5 corresponds to a percent error of 50 if the low value is 30 and a percent error of only 7 if the low value is 0. The percent error is a much better indication of how far the TriBot is from the edge of the line.

setting the range value

The first section of code you need to add computes the value for the Range variable. Figure 18-46 shows the Edit Variables window with the Range and Error variables added. Figure 18-47 shows the blocks for computing the value for the Range variable by subtracting the value of the Low variable from the value of the High variable. These blocks should be placed just after the blocks that compute the target value.

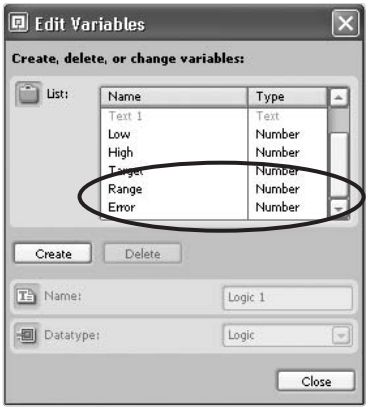


Figure 18-46: Defining the Range variable

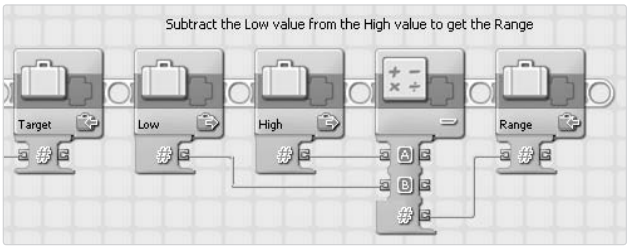


Figure 18-47: Computing the Range variable

Figures 18-48 and 18-49 show the Configuration Panels for the Variable blocks that read the Low and High variables.

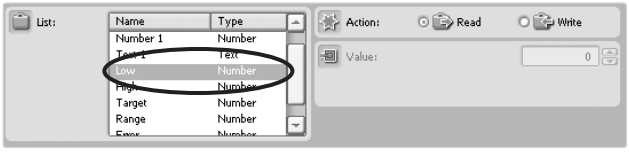


Figure 18-48: Reading the Low variable

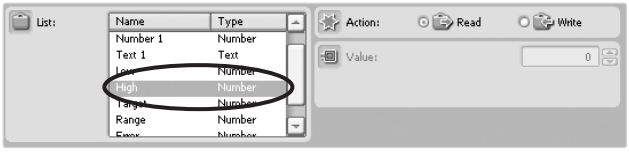


Figure 18-49: Reading the High variable

The Math block subtracts the Low value from the High value. Be sure to correctly connect the data wires between the Variable blocks and the Math block. The Variable block that reads the High value should be connected to the Math block's A data plug, and the Variable block that reads the Low value should be connected to the Math block's B data plug. Figure 18-50 shows the Configuration Panel for the Math block.

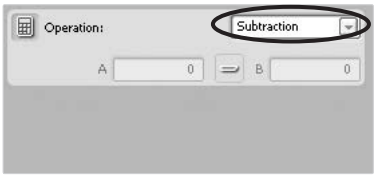


Figure 18-50: Subtracting the Low value from the High value

Figure 18-51 shows the Configuration Panel for the Variable block that stores the result from the Math block in the Range variable.

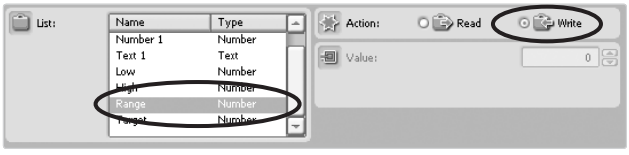


Figure 18-51: Storing the Range value

calculating the error value

The code you've written so far, to read the values from *LF_Config* and calculate the target and range values, is placed before the program's main loop. Once the Target and Range variables have been set, you can start the loop. The first section of code in the loop (shown in Figure 18-52) reads the Light Sensor and calculates the error value using the formula described in "Using Percent Error" on page 245.

Figure 18-53 shows the Configuration Panel for the Light Sensor, which uses all the default settings. If you're using the Color Sensor, use the settings shown in Figure 18-54.

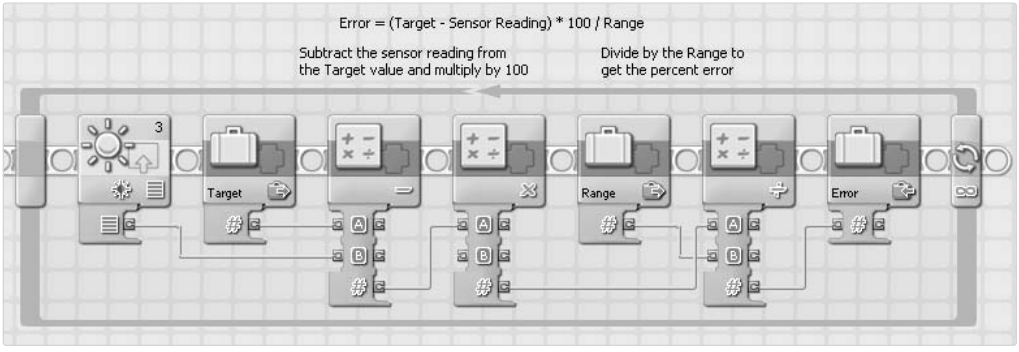


Figure 18-52: Calculating the error value

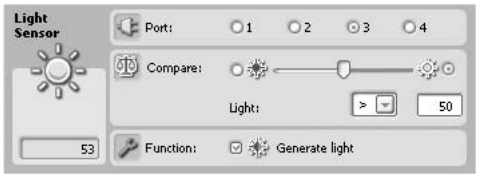


Figure 18-53: Reading the Light Sensor

Figure 18-54: Reading the light level using the Color Sensor

Figure 18-55 shows the Configuration Panel for the Variable block that supplies the target value to the following Math block.

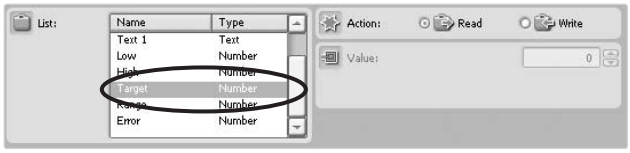


Figure 18-55: Reading the Target variable

The first Math block, shown in Figure 18-56, subtracts the Light Sensor reading from the target value. Be sure to connect the Variable block's Value data plug to the Math block's A data plug and the Light Sensor block's Intensity data plug to the Math block's B data plug. If you reverse these connections, the robot will turn the wrong way.

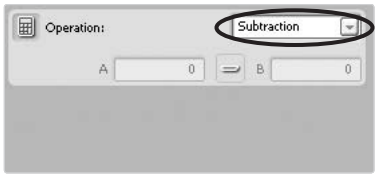


Figure 18-56: Subtracting the Light Sensor reading from the target value

The second Math block, shown in Figure 18-57, multiplies the result from the first Math block by 100. This step allows this program to work with either NXT-G 1.1 or NXT-G 2.0 by ensuring that you'll get a useful number when the next Math block divides the result by the range.

Figure 18-57: Multiplying the result by 100

Figure 18-58 shows the Configuration Panel for the Variable block that reads the Range variable and passes the value to the following Math block.

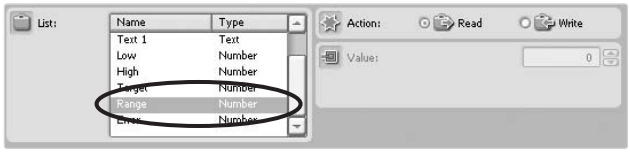


Figure 18-58: Reading the Range value

The next Math block, shown in Figure 18-59, divides the result of the previous Math block by the range to give you the error value as a percent of the total range between the Low and High values.

Figure 18-59: Calculating the error as a percent of the range

Figure 18-60 shows the Configuration Panel for the Variable block that stores the error value.

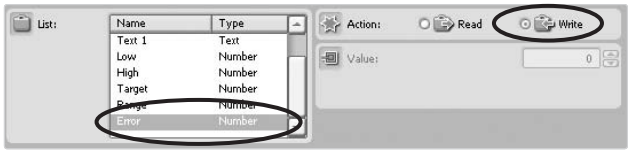


Figure 18-60: Storing the error value

At this point, the program will read the Light Sensor and calculate the error value, which tells you how far the TriBot is from the edge of the line. The sign of the error value (positive or negative) tells you whether the robot is moving away from or toward the center of the line. You'll use the error value to decide how quickly and in which direction the TriBot should turn.

controlling the motors

You have a choice of two blocks for controlling the motors: the Move block and the Motor block. The big advantage of the Move block is that it will synchronize the movements of two motors, allowing you to make the robot move in a straight line, spin in place, or smoothly turn a corner. For example, the WallFollower program uses a Move block to make a 90-degree turn when it reaches an opening in the wall. Once you determine how far the motors need to go to make a quarter turn, you can use a single Move block to control both motors and make the TriBot turn around a corner. The synchronization provided by the Move block works best when the block has enough time to adjust the motors and works less well if the settings are being quickly adjusted in a loop.

You can also control the TriBot using one Motor block for each motor. This has the advantage of giving you more control over exactly how the two motors move, which is especially useful in a program that constantly adjusts the motor settings.

using the motor block

Figure 18-61 shows the Configuration Panel for the Motor block. To control the TriBot, you'll use one Motor block for the B motor and one for the C motor. Controlling how the robot turns is a matter of adjusting the Power settings of the two Motor blocks. If the Power settings of the two blocks are the same, the robot will move in a straight line. If the Power settings are different, the robot will turn left if the B motor is moving faster and right if the C motor is moving faster.

You control how quickly the TriBot turns, and in which direction, by adjusting the Power settings of the two blocks each time through the program's main loop. The arrangement of the two blocks will look like Figure 18-62.

setting the power values

At this point in the discussion we've designed two parts of the program: one part to calculate the error value (shown in Figure 18-52) and the second part, shown in Figure 18-62, to control the motors. The missing piece that ties these two together is the code that uses the error value to calculate the Power values for the two Motor blocks.

understanding the process

The basic idea here is to start with a base power value and then calculate the Power settings for the two Motor blocks by adding and subtracting a steering adjustment based on the error value. The steering adjustment is calculated by multiplying the error value by a value called the *gain*, which determines how much the steering will be adjusted for a given error value. A small gain makes the robot turn slowly and results in very little side-to-side motion, though the robot may not react quickly enough for tight turns. A large gain will allow the TriBot to turn more quickly, but the motion won't be as smooth when the line is fairly straight. Selecting the gain value is called *tuning* the controller and usually involves some trial and error.

Recall that in calculating the error value, you multiplied by 100. After multiplying the error value by the gain, you need to divide the result by 100 to compensate; otherwise, the steering adjustment will be much too large. Listing 18-2 outlines the steps involved in this process.

```
multiply the Error value by the Gain value
divide the result by 100 to get the steering
   adjustment value
set the Power item for the B motor to the base Power
   setting minus the steering adjustment value
set the Power item for the C motor to the base Power
   setting plus the steering adjustment value
```

Listing 18-2: Setting the Power values for the Motor block

Figure 18-63 shows the code for calculating and using the Power settings. The Power variable holds the base power for the program, and you'll initialize this value as well as the Gain variable at the beginning of the program. The

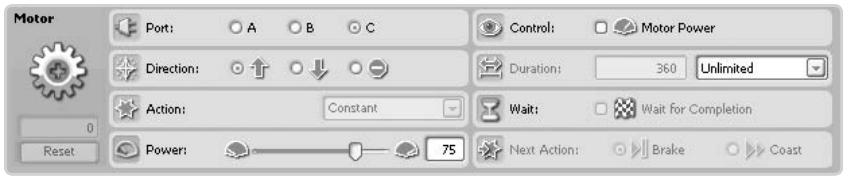


Figure 18-61: The Configuration Panel for the Motor block.

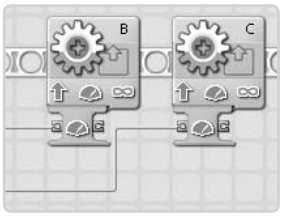


Figure 18-62: Using two Motor blocks to steer the TriBot

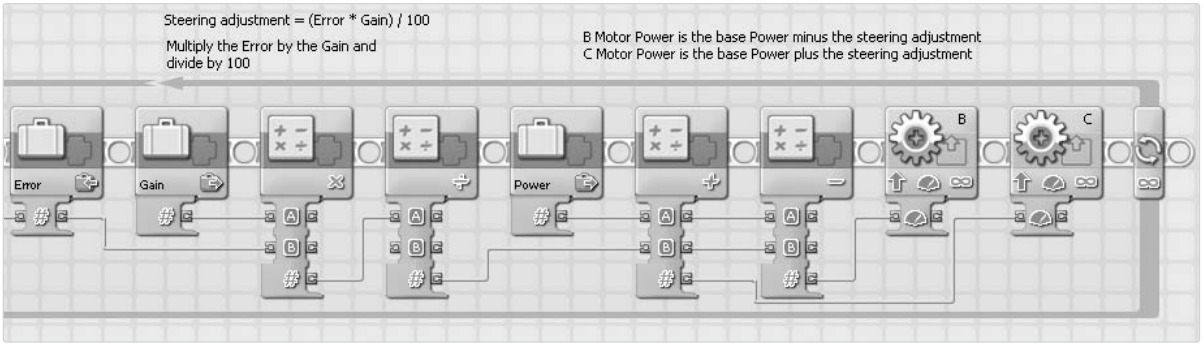


Figure 18-63: Adjusting the steering based on the error value

Power setting for the C motor is calculated by taking the value of the Power variable and adding the steering adjustment value. Similarly, the Power setting for the B motor is calculated by taking the value of the Power variable and subtracting the adjustment value.

a few examples

Let's work through a few examples to make sure this is clear. Say you set Gain to 85 and Power to 75, and the Low and High values collected by the LineFollowerConfig program are 22 and 48. The target value is (High + Low) ÷ 2, which is 35. The Range value is High – Low, which is 26.

Let's see what happens when the TriBot is exactly where you want it to be, directly over the edge of the line, where the Light Sensor reads 35. The percent error value is as follows:

(Target – Sensor Reading) × 100 ÷ Range
(35 – 35) × 100 ÷ 26
0

The steering adjustment is as follows:

(Error × Gain) ÷ 100
(0 × 85) ÷ 100
0

The Power setting for the B motor is as follows:

Power – Steering Adjustment
75 – 0
75

The Power setting for the C motor is as follows:

Power + Steering Adjustment
75 + 0
75

Since the Light Sensor reading matches at the target value, the percent error is 0, which makes the steering adjustment 0. So, both motors will move at the same Power setting (75), and the TriBot will move forward in a straight line.

Now assume the robot is a little to the left of the edge of the line (closer to the center of the line) and the Light Sensor reading is 30. In this case, the percent error value is as follows:

(Target – Sensor Reading) × 100 ÷ Range
(35 – 30) × 100 ÷ 26
19

The steering adjustment is as follows:

(Error × Gain) ÷ 100
(19 × 85) ÷ 100
16

The Power setting for the B motor is as follows:

Power – Steering Adjustment
75 – 16
59

The Power setting for the C motor is as follows:

Power + Steering Adjustment
75 + 16
91

So, for a Light Sensor reading of 30, the B motor power will be set to 59, and the C motor power will be set to 91, which will make the robot turn to the right.

Now, say the TriBot moves to the right, away from the line, and the Light Sensor reads 42. In this case, the percent error is as follows:

(Target – Sensor Reading) × 100 ÷ Range
(35 – 42) × 100 ÷ 26
–26

The steering adjustment is as follows:

(Error × Gain) ÷ 100
(–26 × 85) ÷ 100
–22

When the sensor reading is larger than the target, the error value and the steering adjustment will be negative. All the math still works out properly; you just need to be a little more careful adding and subtracting negative numbers.

The Power setting for the B motor is as follows:

Power – Steering Adjustment
75 – (–22)
97

The Power setting for the C motor is as follows:

Power + Steering Adjustment
75 + (–22)
53

So, for a Light Sensor reading of 42, the B motor power will be set to 97, and the C motor power will be set to 53, which will make the robot turn to the left.

writing the code

Figure 18-63 shows the code for implementing the process described in the previous sections. Now that you understand the process, let's examine the configuration of each of the blocks. Figure 18-64 shows Configuration Panel for the Variable block, which reads the Gain variable.

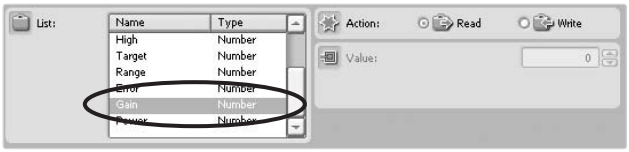


Figure 18-64: Reading the Gain value

Figures 18-65 and 18-66 compute the steering adjustment by multiplying the error value by the Gain value and then dividing by 100.

Figure 18-65: Multiplying the error value and the Gain value

Figure 18-66: Dividing the result by 100

Figure 18-67 shows the Configuration Panel for the next block, which reads the Power variable.

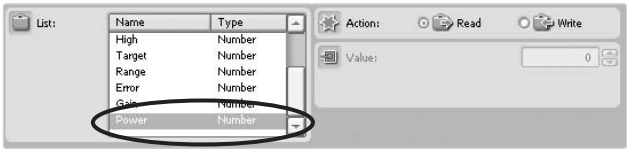


Figure 18-67: Reading the Power variable

The two Math blocks that follow the Variable block both take the value of the Power variable as the A input value and the steering adjustment as the B input value. The first Math block adds the two values, and the second Math block subtracts the steering adjustment from the Power value. Figures 18-58 and 18-69 show the Configuration Panels for these blocks.

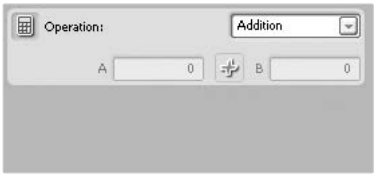


Figure 18-68: Adding steering adjustment to the Power value

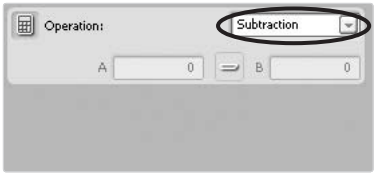


Figure 18-69: Subtracting the steering adjustment from the Power value

Figures 18-70 and 18-71 show the Configuration Panels for the two Motor blocks that use the computed power values to move the robot. The Duration item is set to Unlimited so that the motors will keep moving until the next time through the loop.

Figure 18-70: Moving the B motor

Figure 18-71: Moving the C motor

initializing the gain and power variables

To complete the program, you need to initialize the Power and Gain variables at the beginning of the program, as shown in Figure 18-72.

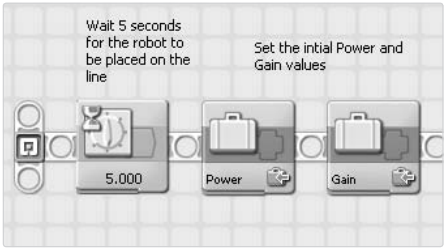


Figure 18-72: Initializing the Power and Gain variables

Set the Power variable to 75 and the Gain variable to 85, as shown in Figures 18-73 and 18-74. These should give you reasonable starting values, which you can adjust during testing to make the TriBot move faster and more smoothly.

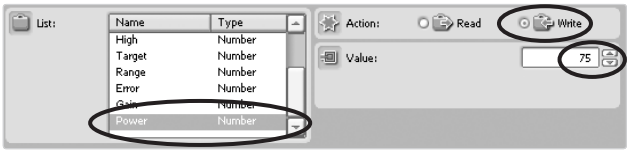


Figure 18-73: Setting the Power variable

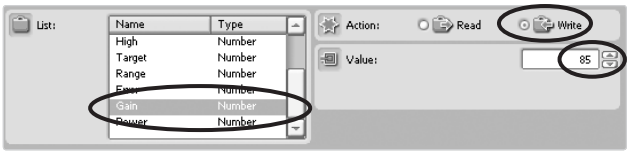


Figure 18-74: Setting the Gain variable

testing the program

Once you have all the blocks configured, try running the program. This program uses a lot of blocks with a lot of connections between them, so it may not work the first time. If there are problems, try using the DisplayNumber block to check the values of the variables. You can also put your data logging skills to the test and try logging the Light Sensor readings and the steering adjustment values to see whether you can spot a problem.

Once you have the program working, try adjusting the Gain and Power values (a little bit at a time) and see how quickly you can make the TriBot move while still staying on the line.

conclusion

Writing a line-following program is a classic NXT exercise that can challenge you to use all your NXT-G skills. The LineFollowerConfig program and the accompanying changes to the LineFollower program from Chapter 6 demonstrate how to use a file to store a setting for a program. This lets you avoid hard-coding values, making the program more flexible. You can use this technique for any program where the sensor target values or other settings may need to be changed.

The LineFollower2 program uses a proportional control algorithm to improve the TriBot's responsiveness to changes in the direction of the line. Using the Light Senor reading and a little math to determine how much the robot should turn allows the robot to both move quicker and stay closer to the line. Using the readings from sensors to control a robot's motors is a basic part of many programs, and experimenting with different control algorithms is a great way to expand your knowledge of robotics while honing your programming abilities.

A

NXT websites

This is a list of websites that have useful information regarding NXT-G programming. Many of these sites also contain links to other online resources that deal with related areas you may be interested in, such as alternate programming languages for the NXT and more general robotic topics.

http://mindstorms.lego.com/ The official LEGO MINSTORMS site contains the latest official NXT news and support information. This site also hosts NXTLOG, a large collection of user-supplied projects.

http://www.legoeducation.com/ The official LEGO Education site provides support to teachers using LEGO products in the classroom.

http://nxtasy.org/ This site provides NXT news, a repository of building instructions and custom-built NXT-G blocks, and links to a wide variety of NXT-related material. The site's message forum provides lots of useful information, including the answers to many common problems that new NXT users encounter. This site is frequented by many knowledgeable NXT users who are very generous with their time, and it is a great place to get your questions answered.

http://www.thenxtstep.blogspot.com/ The NXTStep blog covers news, events, and all things related to NXT. This site also hosts a message forum and is another great place to get questions answered.

http://www.teamhassenplug.org/ This site is maintained by Steve Hassenplug, one of the most experienced NXT users, and contains a wealth of NXT material, including many great building and programming tips.

http://www.nxtprograms.com/ This site has building and programming instructions for a large number of robots based on both the original NXT and NXT 2.0 kits. Working through the programs on this site is a great way to expand your knowledge of NXT-G.

http://www.thenxtclassroom.com/ This site hosts an online community and resources for teachers using the NXT in the classroom. The available material includes building instructions, sample lesson plans, an electronic journal, and links to other online resources.

http://www.legoengineering.com/ This site is supported through a partnership between the Tufts University Center for Engineering Education and Outreach (CEEO) and LEGO Education. The focus of this site is the use of LEGO MINDSTORMS to engage students in Science, Technology, Engineering, and Math (STEM).

B

moving from NXT-G 1.0/1.1 to NXT-G 2.0

This appendix contains information to help you make the transition from NXT-G 1.0/1.1 to NXT-G 2.0. The latest versions of the MINDSTORMS software include some bug fixes and speed improvements, as well as a few changes to the way some blocks behave. The core parts of the NXT-G language have not changed, and the vast majority of the material in this book applies to all versions of the software. There are just a few things you should be aware of to help you use your old programs with your new software.

numbers

The biggest change in NXT-G 2.0 is the switch from integers to floating-point numbers. Although this change won't have any effect on most of the calculations your programs perform, there are two implications that could affect your programs: when dividing two numbers, the result won't be rounded down to an integer, and really big numbers (more than 10 million) may not be exact.

When dividing floating-point numbers, the fractional part of the result is maintained. In general, this makes performing math a bit easier, because you don't need to scale the numbers up to avoid losing important information. Also, the results will more closely match the results you get using a calculator.

Floating-point numbers are only accurate to seven digits. This is only likely to cause problems if you are using very large numbers. For example, you can store the positions of all three motors in a single value by adding the position of the A motor times 1 million, the position of the B motor times 1,000, and the position of the C motor. However, this technique will fail when using floating-point numbers, because the value will be rounded to seven places instead of being the exact value.

For more on the differences between integer and floating-point math, see Chapter 14.

block changes

All the blocks that deal primarily with numbers, such as the Math, Compare, and Range blocks, have been updated to work with floating-point numbers. Beyond those changes, a few blocks have been modified in ways that may be less obvious. Here is a summary of those changes:

* The Math block supports square root and absolute value operations.
* The Number to Text block will show the number rounded the number to two places.
* The File Access block will round a number to two places before writing it to a file.

- The Switch block will round an input value to the nearest integer before matching it against the list of conditions. Even though floating-point numbers are used in NXT-G 2.0, the list of conditions still only supports integer values.
- The Rotation Sensor's Degrees data plug value will be negative when the motor is turned backward.

using old programs

You can use the newer MINDSTORMS software to load, edit, and run any programs you've written using the older software. When you open a program written with the older software, any blocks whose behavior has changed in NXT-G 2.0 will be marked with an exclamation point (!), as shown in Figure B-1. When you run your program, these blocks will work the same way they did with NXT-G 1.0/1.1. The MINDSTORMS software won't automatically change the way your program works; for example, the Math blocks in your old programs won't suddenly start using floating-point division. Changing your blocks automatically could break programs that used to work, which is generally considered a bad thing for a software upgrade to do. Instead of changing your program, the software lets you know which blocks are different in the new version.

Figure B-1: The NXT-G 1.1 Rotation Sensor block as shown in the NXT 2.0 software

After loading an old program with the new software, you can make changes to it. But be very careful if you mix blocks from the two NXT-G versions in the same program—this can cause some very subtle errors, especially when mixing together old and new Math blocks.

side-by-side installation

The education and retail versions of the MINDSTORMS software are installed in different locations on your computer. This allows you to have both the education and retail versions installed at the same time, which is useful if you're using the education set for school and have the retail kit at home. However, when you install the new MINDSTORMS software, the installation process will remove the older version of the software of the same type (that is, the Education 2.0 software will uninstall the Education 1.0/1.1 software, and the Retail 2.0 software will uninstall the Retail 1.0/1.1 software). Uninstalling the older version of the software will remove all the building and programming instructions in the Robo Center, but it will not remove any programs or My Blocks that you've created.

To keep both the old and new versions of the software, follow these instructions when installing the new software:

1. Make a copy of the entire *LEGO Software* folder (found either in the *Applications* folder on a Mac or the *Program Files* folder on a Windows PC), and name the copy something you'll recognize, such as *LEGO Software 1.1*.
2. Install the new MINDSTORMS software.
3. Create a new desktop shortcut to the older version of the software so you can access it easily.

Now you are able to run either version of the software.

index

A

absolute value, 103, 159
adding labels to a displayed value, 108–109
adding a Sequence Beam to a Loop of Switch block, 223–228
algorithm, defined, 86
ambient light, measuring, 63
analyzing data, 210, 214
and, logic operation, 186
AroundTheBlock program, 50, 221–223, 230
art and engineering, 4
assumptions, 88, 233
automatic routing of data wires, 101
A-weighted decibel (dbA), 62

B

backup copies of programs, 11
beep while moving, 71–72
blocks, 5, 9. *See also individual block names*
  connecting with data wires, 97–98
  copying, 137, 198
  in NXT-G 1.1 vs. NXT-G 2.0, 255–256
  running, 73
  selecting 66–67
BlockStartTest program, 229
Bluetooth, 11
Brick. *See* NXT Intelligent Brick
bugs, 12
bumped, Touch Sensor, 58
BumperBot program, 59–61
BumperBot2 program, 71–72, 80
BumperBot3 program, 80–83, 184–185, 186–188
BumperBotWithSound program, 62–63
busy loop, 223
buttons. *See* NXT buttons

C

changing a variable's value, 135–136
Chime My Block, 161–164
clearing the screen. *See* Display block
CoastBack program, 54–44
CoastTest program, 53–55
collecting brightness data, 210–211
Color Lamp block, 227
  Configuration Panel, 227
Color Sensor, 2, 63–64
  Color Sensor mode, 67
  color values, 67
  feedback box, 64, 67
  Light Sensor mode, 64
  View menu, 64
Color Sensor block, 142
  Configuration Panel, 142
  light intensity value, 142
colors, identifying, 64, 67
comma-separated values, 212–214
comments, 13–15
  adding, 14, 51
  comment tool, 14–15
  defined, 13
  program description, 14
  rules for working with, 15
  and the Switch block, 78
common palette, 8
Compare block, 127, 142
  Configuration Panel, 127
compiling, 9
complete palette, 8
computer math, 179
concatenate, 107
conditional, 73
configuration icons, 16
Configuration Panel, 9, 15–16
  changing panels, 15
  disabled items, 16
  general layout, 15
Constant block, 145–146
  choose from list, 145–146
  Configuration Panel, 145
  custom, 146
constants, 144–146
  changing a constant's value, 145–146
  managing, 145
control algorithm, 243
Controller, 9, 11
converting numbers to text, 105
counting objects, 135–136
Create My Block button, 162
Create New Program, 7, 10
Create Pack and Go, 175
crowbar and pin technique, 224–228
custom palette, 8, 163, 174–175

D

data, 97
data collection. *See* data logging
data hub, 99
  opening and closing, 99–101
data logging, 209–220
  collecting sensor data, 210–211
  controlling the amount of data, 216–217
  gaps in the data, 214–216
  timestamp, 212–214
  using LEGO MINDSTORMS Education NXT Software 2.0, 217–220
data plug, 100
  pass-through, 105–106, 109
  restoring pass-through data plugs, 106–107
data types, 104–105
data wires, 97–110
  broken, 109
  cycles, 109
  and data types, 104–105
  deleting, 101
  drawing, 100–101
  and the Loop block, 123–129, 229
  organizing, 157
  routing, 101
  and Sequence Beams, 229
  and starting blocks, 229
  and the Switch block, 111–121, 229
  and time values, 126
dB (decibel), 62
dBA (A-weighted decibel), 62
debugging, 12–13, 95, 208, 237
  steps, 12
  using Sound blocks, 95

decibel (dB), 62
design process, 86–88
detecting
  a clap, 62
  an obstacle, 60, 91, 112
  a person, 69–70
Display block, 11, 152–155
  clearing the screen, 156–158
  controlling image location, 154
  controlling the line setting, 166–167
  drawing, 155
    a circle, 155
    a line, 155, 156–158
  images, 152
  point, 155
  text, 12
  X and Y values, 152, 154
displaying instructions, 235–238
displaying a number, 105, 136
  floating-point numbers, 183
displaying the tone frequency, 105
displaying the volume using text, 117–121
DisplayNumber My Block, 166–174, 205
  building, 167–172
  configuration items, 166, 171–172
  testing, 170
  using, 173, 204, 222, 243, 251
Don't Repeat Yourself (DRY) principle, 138, 206
DoorChime program, 68–70, 161–164, 223–228, 230–232
Download and Run Selected button, 50
downloading a program, 9, 11
drawing. *See* Display block
DRY (Don't Repeat Yourself) principle, 138, 206

E

Edit Constants dialog, 145
Edit Variables dialog, 131–132
edit-compile-test cycle, 13
ending a program. *See* Stop block
engineering, 4
error value, 244
  calculating, 246–247
  percent error, 245
exclusive or (xor), logic operation, 186
Exit button, 59, 147

F

feedback box, 48, 58
File Access block, 195–196, 200, 255
  Configuration Panel, 195–196
  and data types, 196, 200
  operations, 196
FileReader program, 200–201, 237–238
files, 195–208
  closing, 196, 197, 202, 208
  common problems, 208
  common uses for, 195
  creating, 196, 215
  deleting, 196, 197, 207, 208
  downloading, 208
  errors in, 199–200
  gaps in, 214–216
  initial size, 215–216
  managing, 207–208
  naming, 196, 198
  reading, 196, 200, 208, 238
  uploading, 208, 212
  uses, 195
  writing, 196, 197, 200, 208
finding a light source, 140, 141
firmware, 5, 9
  updates, 5
flashing the light, 227–228
floating-point math, 183
  and the Number to Text block, 183
  precision, 183
  range, 183
following a wall, 89–91

G

gain, 248
gears, 45
generating a random number, 184
GentleStop program, 97–101, 112–113
Getting Started window, 7, 11
grouping common settings, 138

H

hard-coded values, 234
hardware, 5
Hello program, 10–11
HelloDisplay program, 11–12
help file, 6, 103
help panel, 9
hertz (Hz), 103
HiTechnic, 3

I

IDE (integrated development environment), 5, 7
initializing the display, 135
initializing variables, 140, 201–204
integer math, 179–180
  division, 180
  order of operations, 180, 181
  range of values, 179
  scaling values, 180, 181
integrated development environment (IDE), 5, 7

K

Keep Alive block, 79–80

L

LabVIEW, 5
Light Sensor, 2, 63–64
  Configuration Panel, 63
  feedback box, 64
  trigger value, 64, 76
  View menu, 64
Light Sensor block, 142, 241
light source, finding, 140, 141
LightPointer program, 138–144
line following, 75–78, 233–251
  control algorithm, 243
  controlling the motors, 248
  error value, 244–247
  how far from the edge, 244–247
  trigger values, 234–235, 239–242
LineFollower program, 75–78, 233–243
LineFollower2 program, 244–251
LineFollowerConfig program, 235–238
Logic block, 185–186
  Configuration Panel, 186
logic operations, 186. *See also individual operations*
logic value, 104
LogicToText progam, 113–116
Loop block, 51–52, 79
  checking the condition at the beginning of the loop, 129
  checking two conditions, 187

Condition data plug, 125
Configuration Panel, 79
control options, 79
Count data plug, 123
and data wires, 123–129
expanding, 226–227
the final loop count value, 124
and multiple Sequence Beams, 229
nested, 124
restarting a loop, 124
setting the loop condition, 125
loop body, 79
loop condition, 79
LoopCountTest program, 123, 229
LoopCountTest2 program, 124
LoopCountTest3 program, 124
LoopStartTest program, 229

M

managing constants, 144–146
managing the custom palette, 174–175
managing memory, 207–208
managing variables, 131–132
magnifying glass tab, 9
manual routing of data wires, 101
math
computer, 179
floating-point, 183
and the Number to Text block, 183
precision, 183
range, 183
integer, 179–180
division, 180
order of operations, 180, 181
range of values, 179
scaling values, 180, 181
Math block, 103, 255
maze, solving, 86–88
following a wall, 89–91
going through an opening, 93–95
testing, 95
turning a corner, 91–93
memory tab, 207
millisecond, 126
mindsensors, 3
MINDSTORMS NXT Kit, 1
building pieces, 2
versions, 3
MINDSTORMS software
sections, 7–9
versions, 3, 255–256
more help link, 9
Motor block, 52–53, 248
comparison with Move block, 53, 248
steering, 248–251
three phases of a motion, 52
motors, 2, 45
power, 45
speed, 45
Move block, 46–48, 248
comparison with Motor block, 53, 248
controlling the Power setting, 97–98
degrees and rotations, 47
determining duration, 49
feedback boxes, 48
following a curve, 50
next action, 48
problem with coasting, 53–55
random duration, 184–185
spinning the robot, 49
synchronized motion, 46
multiple Sequence Beams. *See* Sequence Beams
multitasking, 221
My Block, 161–177. *See also individual My Block names*
broken, 176
Builder window, 162
configuration settings, 164–166, 177
names, 165–166
copying, 175, 230
creating, 161–163
and data plugs, 164–166
adding, 177
and data wires, 164
deleting, 174
editing, 163–164
icons for, 162
moving, 174–175
nested, 176
organizing, 174–175
renaming, 174
and Sequence Beams, 230
sharing, 175
testing, 170
and variables, 176
My Portal, 8
MyBlockTest program, 230

N

National Instruments, 5
Navigation Panel, 9
nested blocks, 78
not, logic operation, 186
Number to Text block, 105, 255
Configuration Panel, 105
and floating-point numbers, 183
NXT Button block, 148
Configuration Panel, 148
NXT buttons, 147
adjusting a value, 150
clearing the screen, 156–158
controlling a loop, 149
pressed vs. bumped, 151–152, 237
NXT Data Logging application, 219–220
NXT Intelligent Brick, 3
NXT-G, 5
moving from 1.1 to 2.0, 255
side-by-side installation,256
NXTSketch program, 155–159

O

Odometer program, 181–182
off-by-one error, 124
old programs, using, 256
online community. *See* websites, NXT-G
or, logic operation, 186
out of memory error, 207

P

percent error, 245
persistence, 195
pixel, 152
Play button, 11
pointer tool, 15
port, 58
PowerSetting program, 148–152, 173
PowerSettingWithImages program, 153–155
precision, 183
program flow, 73
and multiple Sequence Beams, 229–230
programmable timers, 126–129

programming palettes, 8, 9
programming structures, 73
programs. *See also individual program names*
  assumptions, 88, 233
  copying, 175
  downloading, 9
  initial condition, 88
  instructions, 235–238
  practices, 6, 61, 93, 95, 137, 138, 166, 232
  qualities of good, 4
  requirements, 86, 233
  running, 11
  sharing, 174
  testing, 90
  writing, 9
ProgTimer1 My Block, 164–165
prompting the user, 235–238
proportional controller, 244
pseudocode, 83–85

Q

question mark tab, 9

R

Random block, 184
  Configuration Panel, 184, 185
  limits, 184
Range block, 189
reading a program configuration file, 238–241
Record/Play block, 55–56
RedOrBlue program, 64–67, 189–192
RedOrBlueColorMode program, 64–67, 192–194
RedOrBlueCount program, 133–137, 197–199, 201–206
refactoring, 206
reflected light, measuring, 63
remembering, a position, 140
remote control tool, 56
requirements, 4
  for programs, 86, 233
Reset Motor block, 55
returning to a position, 143–144
right-hand rule algorithm, 86
Robo Center, 8
Rotation Sensor, 2, 45, 70–71
  Configuration Panel, 70
Rotation Sensor block, 70–71
  NXT-G 1.1 vs. NXT-G 2.0, 103, 158–159, 256

S

saving your work, 10
scaling values, 180
Sensor blocks, 57, 99, 113
  advantages of, 113
sensors, 2, 57. *See also individual sensor names*
  selecting a port, 61
Sequence Beams, 9, 73
  adding, 221–223, 225–226
  keeping out of trouble, 232
  multiple, 221–232
  and My Blocks, 230
  and program flow, 229–230
  synchronizing, 230–232
side-by-side installation, 256
SimpleMove program, 46
simplifying assumption, 64
sleep timer, 79
software, 5
Sound block, 10, 69, 103
  controlling tone with, 103–104
  controlling volume with, 102, 117
  and debugging, 95
  playing sound file with, 10
  playing tone with, 69
Sound Sensor, 2, 61–62
  Configuration Panel, 61
  trigger value, 61
  View menu, 62
SoundMachine program, 101–109, 117–121
source code, 9, 83
  for example programs, 6
square root, 103
stall, motor, 47
Start Data Logging block, 217–218
Start New Program. *See* Create New Program
steering, 47–48
Stop block, 80
Stop Data Logging block, 217–218
storing
  program data, 195, 197–199, 201–205
  program settings, 195, 235–238
Switch block, 64–65, 73–75, 255
  and comments, 78
  conditions list, 116–117
    adding a condition, 116,
    the default condition, 117, 119
    fixing the order, 119–120
    removing a condition, 116
  Configuration Panel, 74–75
  and data types, 111
  and data wires 111–121
    connecting data wires, 111, 114–116, 120
    input data plug, 111
    multiple Sequence Beams, 229
    passing data in, 113
    passing data out, 113–114
  Flat and Tabbed View, 74, 78, 113
  more than two choices, 76, 116–117
  nested, 77
  numbers as input, 117
  NXT-G 1.1 vs. NXT-G 2.0, 117
  tabs, 74, 121
  trigger, 74
  value control, 111

T

target value, 239
Temperature Sensor, 3
testing, 13, 52, 93, 137, 242
  single block, 50
Text block, 107
  combining values, 212–214
  Configuration Panel, 107
text value, 104
ThereAndBack program, 49–50
three-state controller, 243
time values, 126
Timer block, 125–126
  Configuration Panel, 125
  range of values, 126
  trigger, 126
Timer1 program, 126
Timer2 program, 127
Timer3 program, 129
timers, 125

timestamp, 212
Touch Sensor, 2, 58
  Configuration Panel, 58
  feedback box, 58
  View menu, 59
  waiting for the user, 66
Touch Sensor block, 114
TriBot, building instructions, 17
  alternate placement for Color Sensor, 42
  alternate placement for Ultrasonic Sensor, 43
  wires, 41
trigger, 58, 74, 234–235
truth table, 186
tuning a controller, 248

U

Ultrasonic Sensor, 2, 68
  Configuration Panel, 68
  feedback box, 68
  inches and centimeters, 99
  range, 68
  trigger, 68, 69
  View menu, 68
Ultrasonic Sensor block, 99
undo, 10
USB, 11

V

Variable block, 132
  Configuration Panel, 132
variables, 131–144, 243
  creating, 131–132, 133–134
  and data types, 132
  deleting, 132
  in place of long data wires, 138, 243
  initializing, 140, 201–204
  and My Blocks, 176
  naming, 132, 133
VerifyLightPointer program, 209–217
VerifyLightPointer2 program, 218–219
Vernier, 3
View menu, 48, 59

W

Wait block
  Sensor, 57
  Time, 13
Wait for Completion, 13
WallFollower program, 83–96
websites, NXT-G, 2, 253
wheel circumference, 181
wheels as dials, 101–102, 155–158
  NXT-G 2.0, 158
whole numbers. *See* integer math
work area, 8

X

xor. *See* exclusive or (xor), logic operation

More no-nonsense books from **no starch press**

The LEGO® MINDSTORMS® NXT 2.0 Discovery Book

A Beginner's Guide to Building and Programming Robots

by LAURENS VALK

The crystal-clear instructions in *The LEGO MINDSTORMS NXT 2.0 Discovery Book* show you how to harness the capabilities of the NXT 2.0 set to build and program your own robots. Author and robotics instructor Laurens Valk walks you through the set, showing you how to use its various pieces and how to use the NXT software to program robots. Interactive tutorials make it easy for you to reach an advanced level of programming as you learn to use data wires and variables, monitor sensors, and build robots that move. You'll build eight increasingly sophisticated robots like the Strider (a six-legged walking creature), the CCC (a climbing vehicle), and the Hybrid Brick Sorter (a robot that sorts by color and size). Numerous building and programming challenges throughout the book encourage you to think creatively and apply what you've learned as you develop the skills essential to creating your own robots.

MAY 2010, 336 PP., $29.95
ISBN 978-1-59327-211-1

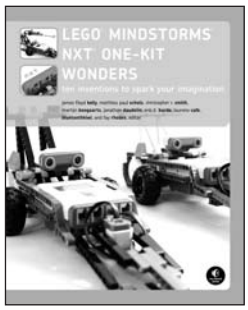
LEGO® MINDSTORMS® NXT Thinking Robots

Build a Rubik's Cube Solver and a Tic-Tac-Toe Playing Robot!

by DANIELE BENEDETTELLI

Daniele Benedettelli is world famous for his innovative LEGO MINDSTORMS robots. His YouTube videos have been viewed more than two million times, and his robots have been featured on international television programs and all over the Web. *LEGO MINDSTORMS NXT Thinking Robots* includes full building and programming instructions for two of Benedettelli's most unique creations—a brand-new version of his famous Rubik's Cube solver and an interactive Tic-Tac-Toe playing robot. You will find complete building instructions for each robot—whether you are using the NXT 2.0 set or the original NXT 1.0. Benedettelli includes information about how to use the robots as well as explanations of the artificial intelligence that makes these robots think.

DECEMBER 2009, 224 PP., $29.95
ISBN 978-1-59327-216-6



LEGO® MINDSTORMS® NXT One-Kit Wonders

Ten Inventions to Spark Your Imagination

by JAMES FLOYD KELLY, MATTHIAS PAUL SCHOLZ, CHRISTOPHER R. SMITH, MARTIJN BOOGAARTS, JONATHAN DAUDELIN, ERIC D. BURDO, LAURENS VALK, BLUETOOTHKIWI, *and* FAY RHODES

LEGO MINDSTORMS NXT One-Kit Wonders is packed with building and programming instructions for ten innovative robots. The book dives headfirst into the creative thrill of robot-building with models like Grabbot, Dragster, and The Hand. Step-by-step building instructions make it simple to construct even the most complex models, while the detailed programming instructions teach you how a NXT program really works. And best of all, you only need one NXT Retail kit to build all ten of the robots!

NOVEMBER 2008, 408 PP., $29.95
ISBN 978-1-59327-188-6

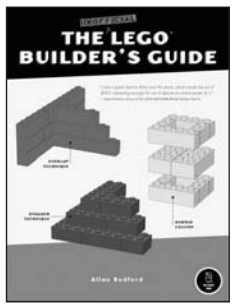
Forbidden LEGO®

Build the Models Your Parents Warned You Against!

by ULRIK PILEGAARD *and* MIKE DOOLEY

Written by a former master LEGO designer and a former LEGO project manager, this full-color book showcases projects that break the LEGO Group's rules for building with LEGO bricks—rules against building projects that fire projectiles, require cutting or gluing bricks, or use nonstandard parts. Many of these are back-room projects that LEGO's master designers build under the LEGO radar, just to have fun. Learn how to build a catapult that shoots M&Ms, a gun that fires LEGO beams, a continuous-fire ping-pong ball launcher, and more! Tips and tricks will give you ideas for inventing your own creative model designs.

AUGUST 2007, 192 PP., *full color*, $24.95
ISBN 978-1-59327-137-4



The Unofficial LEGO® Builder's Guide

by ALLAN BEDFORD

The Unofficial LEGO Builder's Guide combines techniques, principles, and reference information for building with LEGO bricks that go far beyond LEGO's official product instructions. You'll discover how to build everything from sturdy walls to a basic sphere, as well as more advanced structures, including a mini space shuttle and a train station. The book also delves into advanced concepts such as scale and design. The book covers essential terminology and includes the Brickopedia, a comprehensive guide to the different types of LEGO pieces.

SEPTEMBER 2005, 344 PP., $24.95
ISBN 978-1-59327-054-4

PHONE:
800.420.7240 OR
415.863.9900
MONDAY THROUGH FRIDAY,
9 AM TO 5 PM (PST)

FAX:
415.863.9950
24 HOURS A DAY,
7 DAYS A WEEK

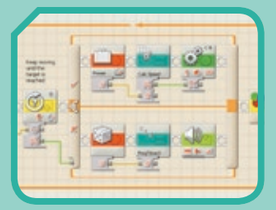
EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

updates

Visit *http://www.nostarch.com/nxt-g.htm* for updates, errata, and other information.

The Art of LEGO MINDSTORMS NXT-G Programming is set in Chevin. The book was printed and bound at Malloy Incorporated in Ann Arbor, Michigan. The paper is 60# Spring Forge, which is certified by the Sustainable Forestry Initiative (SFI). The book has a RepKover binding, which allows it to lie flat when open.



THE COMPLETE GUIDE TO PROGRAMMING WITH NXT-G

The LEGO® MINDSTORMS® software and its NXT-G programming language are powerful tools that make it easy to write custom programs for your robots. NXT-G is a great first programming language, but that doesn't mean it's easy to understand—at least not right away.

In *The Art of LEGO MINDSTORMS NXT-G Programming*, author and experienced software engineer Terry Griffin explains how to program MINDSTORMS robots with NXT-G. You'll learn how to work with the core parts of the NXT-G language, such as blocks, data wires, files, and variables, and see how these pieces can work together. You'll also learn good programming practices, bad habits to avoid, and useful debugging strategies.

As you follow along with the book's extensive instructions and explanations, you'll learn exactly how NXT-G works and how to:

* Write custom programs that make your robots appear to think and respond to your commands
* Design, create, and debug large programs
* Write programs that use data wires and the NXT buttons to turn a robot into a contraption, like a sound generator or a sketch pad
* Use My Blocks in your programs, and share them with others
* Store data on the NXT, manage its memory, and transfer files between the NXT and your computer

The book's programs work with one general-purpose test robot that you'll build in Chapter 3.

Whether you're a young robotics enthusiast, an adult working with children to learn robotics, a parent, a FIRST LEGO League coach, or a teacher using NXT in the classroom, this is the complete guide to NXT-G that you've been looking for.

about the author

Terry Griffin has been a software engineer for over 20 years, spending most of his time creating software for controlling various types of machines. He works for Carl Zeiss SMT on the Orion Helium Ion Microscope, programming the user interface and high-level control software. He lives in Massachusetts with his wife, Liz, a middle school math and science teacher, and their three daughters, Cheyenne, Sarah, and Samantha.

REQUIREMENTS: ONE LEGO MINDSTORMS NXT OR NXT 2.0 SET

no starch press

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

RepKover
"I LIE FLAT."
This book uses RepKover—a durable binding that won't snap shut.

PRICE: *$29.95 ($37.95 CDN)* **SHELVE IN:** *ROBOTICS/HOBBIES*

ISBN: 978-1-59327-218-0
52995
9 781593 272180
6 89145 72189 8

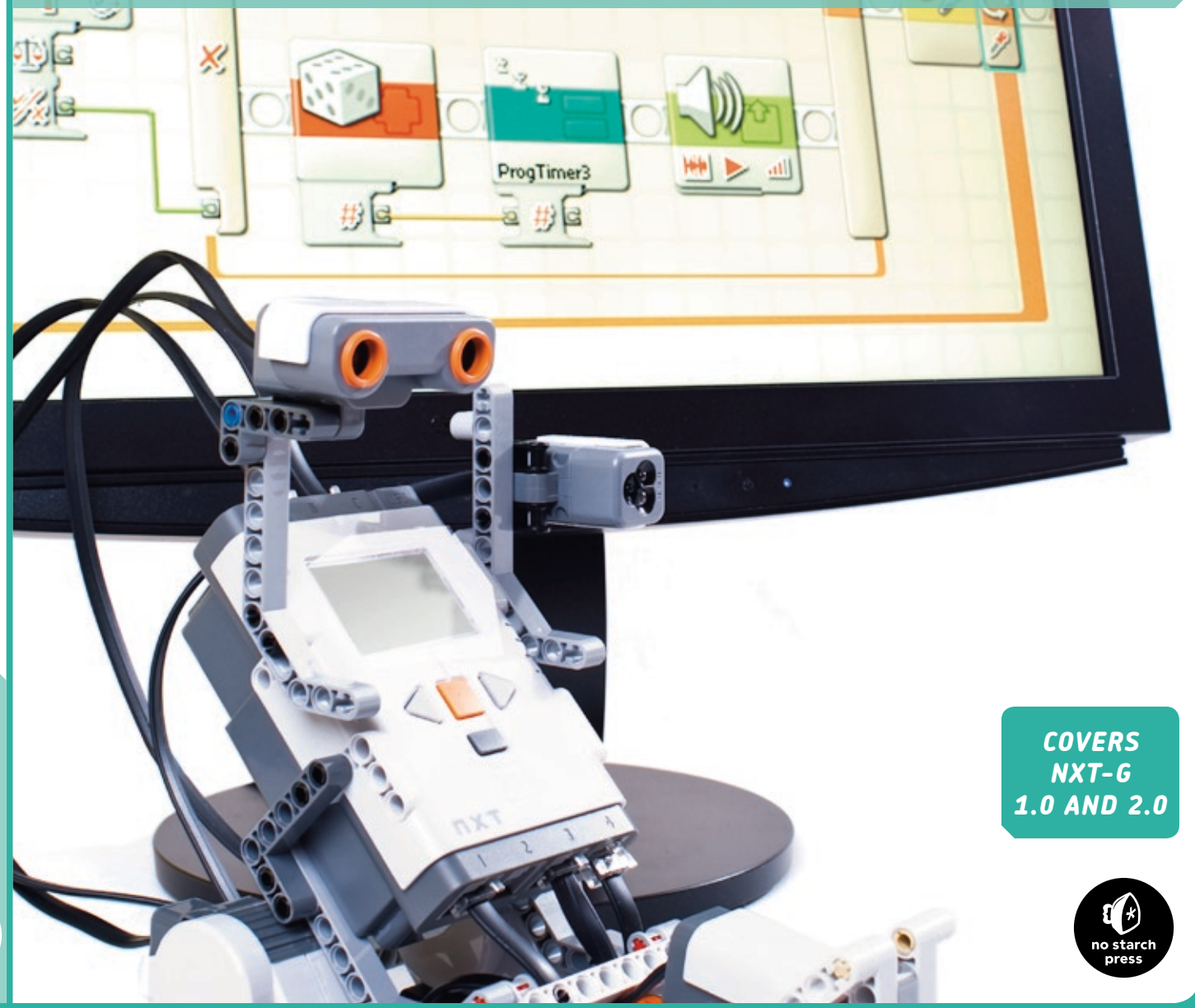
THE ART OF LEGO® MINDSTORMS® NXT-G PROGRAMMING

griffin

no starch press

THE ART OF LEGO® MINDSTORMS® NXT-G PROGRAMMING

terry **griffin**



COVERS NXT-G 1.0 AND 2.0

no starch press